

(2)

AFOOK-TK. 89-0731

AD-A208 739

THE PASM PARALLEL PROCESSING SYSTEM:
DESIGN, SIMULATION, AND IMAGE PROCESSING APPLICATIONS

Volume I

A Thesis

Submitted to the Faculty

of

Purdue University

by

James T. Kuehn



In Partial Fulfillment of the
Requirements for the Degree

of

Doctor of Philosophy

May 1986

| |
|-----------------------------|
| DISTRIBUTION STATEMENT A |
| Approved for public release |
| Distribution Unlimited |

89 6 06 003

REPORT DOCUMENTATION PAGE

UNCLASSIFIED

| | | | |
|---|-------|--|---|
| 1a. REPORT SECURITY CLASSIFICATION | | 1b. RESTRICTIVE MARKINGS | |
| | | 3. DISTRIBUTION/AVAILABILITY OF REPORT <i>Approved for public release, distribution unlimited</i> | |
| 2a. SECURITY CLASSIFICATION AUTHORITY | | 4. PERFORMING ORGANIZATION REPORT NUMBER(S) | |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | | | |
| 6a. NAME OF PERFORMING ORGANIZATION PURDUE UNIVERSITY | | 6b. OFFICE SYMBOL <i>(If applicable)</i> | 5. MONITORING ORGANIZATION REPORT NUMBER(S) <i>AFOSR - TR - 89 - 0731 F49620-86-K-0006</i> |
| 6c. ADDRESS (City, State, and ZIP Code) SCHOOL OF ELECTRICAL ENGINEERING WEST LAFAYETTE, IN 47907 | | 7a. NAME OF MONITORING ORGANIZATION AFOSR/NE | |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFOSR | | 8b. OFFICE SYMBOL <i>(If applicable)</i> NE | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER <i>F49620-86-K-0006</i> |
| 8c. ADDRESS (City, State, and ZIP Code) BUILDING 410 BOLLING AFB, DC 20332 | | 10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 61102F | |
| 11. TITLE (Include Security Classification) THE MAPPING OF PARALLEL ALGORITHMS TO RECONFIGURABLE PARALLEL ARCHITECTURES/CONTRIBUTED PAPERS 36/AN APPLICATION OF TENSOR THEORY TO 3-D SHAPE ANALYSIS/ THE PASM PARALLEL PROCESSING SYSTEM: DESIGN, SIMULATION, AND IMAGE PROCESSING APPLICATIONS/ | | PROJECT NO. DARPA | TASK NO. |
| 12. PERSONAL AUTHOR(S) | | WORK UNIT ACCESSION NO. | |
| 13a. TYPE OF REPORT FINAL | | 13b. TIME COVERED FROM 01JAN86 TO 31DEC89 | 14. DATE OF REPORT (Year, Month, Day) |
| 16. SUPPLEMENTARY NOTATION | | 15. PAGE COUNT | |
| 17. COSATI CODES | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) | |
| FIELD | GROUP | SUB-GROUP | |
| | | | |
| | | | |
| 19. ABSTRACT (Continue on reverse if necessary and identify by block number) | | | |
| This Final Report DARPA Contract No. F49620-86-K-0006 is in the form of published papers, a masters thesis, a Ph.D. thesis, and a Ph.D. thesis proposal all supported in whole or part by the contract. | | | |
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS | | 21. ABSTRACT SECURITY CLASSIFICATION <i>UNCLASSIFIED</i> | |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL <i>Guis</i> | | 22b. TELEPHONE (Include Area Code) <i>(303) 777-4431</i> | 22c. OFFICE SYMBOL <i>NE</i> |

89 600 005

ACKNOWLEDGEMENTS

I gratefully acknowledge the support and guidance of my thesis advisor and friend, Professor Howard Jay Siegel, who has taught me at least three important lessons: that deadlines shall never be considered cast in stone, that the number of local reference citations shall never exceed one-third the total, and that interconnection networks aren't really so bad after all.

The members of the advisory, oral preliminary examination, and final oral examination committees: Professors Edward Delp, Dennis B. Gannon, Piyush Mehrotra, David G. Meyer, Leah H. Jamieson, and Philip H. Swain are also acknowledged for their guidance and efforts.

Sincere thanks go to Thomas Schwederski who has the special talent of turning ideas into hardware reality. Working with him has always been especially enjoyable and fruitful.

Discussions with Jose Fortes, Bill Carlson, Nat Davis, George Adams, Dave Tuomenoksa, and Tom Rice have been most useful. Finally, there are the dozens of graduate and undergraduate students who have contributed their time and efforts toward the PASM design, system software, simulator and application algorithms. They are: Troy Cauble, Henry Choy, Heuey Cheung, Jeff Conrad, Dave Czenkusch, Tom Dungan, Todd Eberwine, Jeff Fessler, Rick Heidebrecht, Craig Hughes, Lee Jesionowski, Tim Johnson, Brent Kaser, Robert King, Bob Klose, Ray Lang, Louie Lau, Andy Mayer, Charles Mok, John Rozwat, Karen Russell, Ken Saunders, Thomas Schwederski, and Tjun Wong. I appreciate all of their efforts.

Work in this thesis was supported by the Defense Mapping Agency monitored by the United States Air Force, Rome Air Development Center, Information Sciences Division under Contract No. F30602-78-0025 through the University of Michigan; by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under Grant No. AFOSR-78-3581; by the United

States Army Research Office, Department of the Army, under grant number DAAG29-82-K-0101; by the Defense Mapping Agency monitored by the United States Air Force Command, Rome Air Development Center, through the Southeastern Center for Electrical Engineering Education under contract number F30602-C-0193; by the United States Air Force Command, Rome Air Development Center, under contract number F30602-83-K-0119; by an IBM Graduate Fellowship; and by the Air Force Office of Scientific Research under Grant No. F49620-86-K-0006.

A-1



TABLE OF CONTENTS

| | Page |
|--|-------------|
| LIST OF TABLES | viii |
| LIST OF FIGURES | ix |
| ABSTRACT | xiii |
| GLOSSARY OF TERMS AND SYMBOLS | xv |
| THESIS MATERIAL PREVIOUSLY PUBLISHED BY THE AUTHOR..... | xix |
| PART I PASM AND THE PASM PROTOTYPE..... | 1 |
| CHAPTER 1 - INTRODUCTION..... | 2 |
| CHAPTER 2 - PARALLEL COMPUTATION MODELS | 3 |
| 1.2.1 SIMD Model..... | 3 |
| 1.2.2 MIMD Model | 5 |
| 1.2.3 MSIMD and MMIMD Models | 7 |
| 1.2.4 Pipeline Model..... | 7 |
| 1.2.5 Hybrids | 8 |
| 1.2.6 Performance Characteristics | 9 |
| 1.2.7 Interconnection Structures..... | 11 |
| 1.2.8 Masking Schemes | 37 |
| CHAPTER 3 - SURVEY OF RELATED LITERATURE | 39 |
| 1.3.1 SIMD Machines..... | 40 |
| 1.3.2 MIMD Machines..... | 43 |
| 1.3.3 MSIMD and MMIMD Machines | 48 |
| 1.3.4 SIMD/MIMD Machines | 48 |
| CHAPTER 4 - PASM OVERVIEW..... | 50 |
| 1.4.1 Basic Components..... | 50 |
| 1.4.2 Parallel Computation Unit | 50 |
| 1.4.3 MCs and Partitions..... | 53 |
| 1.4.4 Secondary Storage | 55 |
| 1.4.5 Memory Management System..... | 55 |
| 1.4.6 System Control Unit..... | 56 |

| | Page |
|---|------------|
| CHAPTER 5 - PASM HISTORY | 57 |
| CHAPTER 6 - PASM DESIGN STUDIES..... | 61 |
| 1.6.1 Functional Description of System Components | 62 |
| 1.6.2 PASM PE Design | 65 |
| 1.6.3 PASM Interconnection Network and Interface | 83 |
| 1.6.4 PASM MC Design | 89 |
| 1.6.5 Overlapping Schemes | 117 |
| 1.6.6 PASM Secondary Memory Systems | 120 |
| 1.6.7 PASM System Control and Memory Management | 122 |
| CHAPTER 7 - PASM PROTOTYPE DESIGN..... | 123 |
| 1.7.1 Role of the Author..... | 123 |
| 1.7.2 General Design Constraints and Guidelines | 124 |
| 1.7.3 Prototype Implementation Summary | 126 |
| CHAPTER 8 - LESSONS FROM THE PROTOTYPE | 148 |
| CHAPTER 9 - TOWARD A 1024-PE PASM SYSTEM..... | 151 |
| 1.9.1 Performance Goals | 151 |
| 1.9.2 Hardware Constraints | 152 |
| 1.9.3 1024-PE PASM Proposal | 153 |
| 1.9.4 Summary | 166 |
| CHAPTER 10 - SUMMARY | 168 |
| PART II PARALLEL LANGUAGES AND OPERATING SYSTEMS | 169 |
| CHAPTER 1 - INTRODUCTION | 170 |
| CHAPTER 2 - SUMMARY OF PASM LANGUAGE AND OPERATING SYSTEM APPROACHES | 171 |
| 2.2.1 The Case for Parallel Languages | 171 |
| 2.2.2 Language Characteristics | 172 |
| 2.2.3 Distributed Operating Systems | 174 |
| CHAPTER 3 - PARALLEL ASSEMBLER AND SUPPORT SOFTWARE | 179 |
| 2.3.1 Prototype Assembly Language and Assembler | 179 |
| 2.3.2 Program Sections | 180 |
| 2.3.3 Prototype Parallel Assembler Criticisms | 183 |
| 2.3.4 Support Software Conversion Experiences..... | 184 |
| CHAPTER 4 - PARALLEL-C | 187 |
| 2.4.1 Introduction | 187 |
| 2.4.2 Goals and Philosophies | 188 |
| 2.4.3 Extensions for SIMD Mode | 190 |
| 2.4.4 Extensions for MIMD Mode | 218 |
| 2.4.5 Compilation Techniques for PASM..... | 222 |

| | Page |
|---|------------|
| 2.4.6 Programming Examples..... | 236 |
| 2.4.7 Syntax Summary | 238 |
| 2.4.8 Summary..... | 244 |
| CHAPTER 5 - PASM OPERATING SYSTEM STUDIES | 245 |
| 2.5.1 Introduction | 245 |
| 2.5.2 Memory Management System Overview..... | 246 |
| 2.5.3 Handling Large Data Sets..... | 250 |
| 2.5.4 Distributing Files Among Storage Devices..... | 251 |
| 2.5.5 Memory Management System Processors..... | 253 |
| 2.5.6 PASM Prototype Operating System Hierarchy..... | 259 |
| 2.5.7 Summary..... | 271 |
| CHAPTER 6 - SUMMARY..... | 272 |
| PART III PARALLEL IMAGE PROCESSING ALGORITHM STUDIES ... | 273 |
| CHAPTER 1 - INTRODUCTION | 274 |
| CHAPTER 2 - DEVELOPMENT AIDS | 275 |
| CHAPTER 3 - PERFORMANCE MEASURES | 279 |
| 3.3.1 Introduction | 279 |
| 3.3.2 Performance Measures | 279 |
| 3.3.3 Summary..... | 288 |
| CHAPTER 4 - SMOOTHING ALGORITHM..... | 289 |
| CHAPTER 5 - HISTOGRAMMING ALGORITHM..... | 302 |
| CHAPTER 6 - CONTOUR EXTRACTION | 311 |
| 3.6.1 Introduction | 311 |
| 3.6.2 Edge-Guided Thresholding..... | 312 |
| 3.6.3 Contour Tracing | 314 |
| 3.6.4 Architectural Implications..... | 319 |
| CHAPTER 7 - RASTER-TO-VECTOR CONVERSION..... | 325 |
| 3.7.1 Introduction | 325 |
| 3.7.2 Line Thinning | 326 |
| 3.7.3 Parallel Vectorization..... | 335 |
| 3.7.4 Summary of Results..... | 344 |
| CHAPTER 8 - VECTOR-TO-RASTER CONVERSION..... | 347 |
| CHAPTER 9 - SUMMARY..... | 354 |
| SUMMARY OF CONTRIBUTIONS | 355 |
| LIST OF REFERENCES..... | 357 |

| | Page |
|--|-------------|
| APPENDICES | |
| APPENDIX A1 - MANUALS..... | 372 |
| A1.1 Introduction to 68000-Family Utilities..... | 373 |
| A1.2 Pa68 Assembler Reference Manual | 381 |
| A1.3 Cc68 Compiler Calling Conventions Guide..... | 431 |
| A1.4 PASM Parallel Processing Laboratory Manual Set..... | 440 |
| A1.5 PASM - XINU Notes File | 460 |
| A1.6 PSST Simulator Manual | 475 |
| APPENDIX A2 - DEVELOPMENT AIDS..... | 533 |
| A2.1 Parallel Assembler Source Code..... | 534 |
| A2.2 PSST Simulator Source Code | 744 |
| A2.3 PSET Standard Library Source Code | 832 |
| APPENDIX A3 - IMAGE PROCESSING SOFTWARE | 921 |
| A3.1 Smoothing Algorithms | 922 |
| A3.2 Histograming Algorithms | 941 |
| A3.3 Edge-Guided Thresholding Algorithm | 947 |
| A3.4 Raster-to-Vector Conversion Algorithms | 952 |
| A3.5 Vector-to-Raster Conversion Algorithm | 996 |
| VITA | 1065 |

LIST OF TABLES

| Table | Page |
|---|------|
| 1.8.1. Physical board requirements using prototype implementation technology. | 149 |
| 2.4.1. Program sections for Parallel-C variables. | 223 |
| 3.4.1. Comparison of smoothing algorithm simulation and timing characteristics. The "original" algorithm run time results are normalized to 1.00. The internal cycle time is 250ns. All of the simulations are performed with N = 16 PEs. | 297 |
| 3.4.2. Smoothing simulation SIMD speedup, efficiency, and utilization results. | 300 |
| 3.5.1. Histograming simulation SIMD execution time results. The total image size varies from 4-by-4 to 64-by-64 and is distributed over N = 16, 64, and 256 PEs. | 305 |
| 3.5.2. Histograming simulation SIMD speedup, efficiency, and utilization results. | 308 |
| 3.7.1. Thinning simulation all-SIMD and part-SIMD part-MIMD execution time results. | 346 |
| 3.8.1. Vector-to-raster conversion algorithm simulation results for 2713 vectors and a varying number of PEs. | 353 |
| A1.6.1. Utility programs needed to support PSST in a VAX/UNIX environment. | 479 |
| A1.6.2. Cross-development software for the 68000 family of CPUs. | 479 |

LIST OF FIGURES

| Figure | Page |
|--|------|
| 1.2.1. SIMD/MIMD machine model (PE-to-PE configuration). | 3 |
| 1.2.2. SIMD/MIMD machine model (P-to-M configuration). | 5 |
| 1.2.3. Bus network for $N=8$. | 14 |
| 1.2.4. Shared memory connection for $N=8$. | 16 |
| 1.2.5. Complete interconnection network for $N=8$. | 17 |
| 1.2.6. Unidirectional ring network for $N=8$. | 18 |
| 1.2.7. Bidirectional ring network for $N=8$. | 19 |
| 1.2.8. Four-nearest-neighbor (Illiad) connection for $N=16$. | 21 |
| 1.2.9. Eight-nearest-neighbor connection for $N=16$. | 22 |
| 1.2.10. Pyramid connection for $H=3$, $b=4$. | 23 |
| 1.2.11. Perfect shuffle network for $N=8$. | 25 |
| 1.2.12. PM2I network for $N=8$. | 26 |
| 1.2.13. Cube network for $N=8$. | 28 |
| 1.2.14. (a) Multistage Generalized Cube topology, shown for $N=8$. (b) Example one-to-one connection (input 2 to output 4). (c) Example broadcast connection (input 5 to outputs 2, 3, 6, and 7). (d) Example permutation connection (input i to out- put $i+1$ modulo N). | 29 |
| 1.2.15. Extra Stage Cube network for $N=8$. | 32 |
| 1.2.16. Data Manipulator network for $N=8$. | 33 |
| 1.4.1. PASM block diagram. | 51 |

| Figure | Page |
|---|------|
| 1.4.2. PASM prototype overview. | 52 |
| 1.6.1. MC organization consisting of a combined MC CPU/FBU and combined MC CPU/FBU memory. | 92 |
| 1.6.2. MC organization consisting of a combined MC CPU/FBU and combined MC CPU/FBU memory. A separate Instruction Broadcast Bus reduces traffic on the MC Bus. | 94 |
| 1.6.3. Master FBU - slave MC CPU organization. | 96 |
| 1.6.4. Master MC CPU - slave FBU organization. | 100 |
| 1.6.5. Internal master/slave FBU organization. | 102 |
| 1.6.6. MC global status communication tree. | 111 |
| 1.6.7. (a) Reconfigurable shared bus scheme for interconnecting MC processors and memory modules, shown for Q=8, where each box can be set to "through" or "short." (b) Bus set for MC 0, 2, 4, and 6 forming one machine partition, MCs 1 and 5 forming a second machine partition, MC 3 forming a third machine partition, and MC 7 forming a fourth machine partition. | 114 |
| 1.7.1. CPU, Memory, PE-Network Interface, and MC-PE I/O Boards. DTRin and DTRout are connected to the network input and output, respectively. B and N refer to the "bypass" and "normal" network DTR inputs and outputs. | 128 |
| 1.7.2. Network Input, Intermediate, and Output Stage Boards. UI and UO are the "normal" upper input and output, respectively. Similarly, LI and LO are the lower input and output. UIB, UOB, LIB, and LOB are the corresponding "bypass" connections. | 129 |
| 1.7.3. Parallel Port and Fetch Unit Boards. | 130 |
| 1.7.4. Disk Access Switch, Disk Controller (MVME-320), Partition Combination, and System Controller Boards (MVME-050). | 131 |
| 1.7.5. 2-by-2 crossbar switch controlled by signals C ₁₋₄ . UI and UO are the upper input and output, respectively. Similarly, LI and LO are the lower input and output. | 138 |
| 1.9.1. PE organization. | 154 |
| 1.9.2. PE board layout for a 1024-PE PASM system. | 156 |

| Figure | | Page |
|---------|--|------|
| 1.9.3. | Physical organization of a PASM quadrant containing 256 PEs arranged in 8 MC groups. | 157 |
| 1.9.4. | MSU structure. | 159 |
| 1.9.5. | 2-by-2 network interchange box. The input Handshake lines are comprised of a request signal from and a grant signal to stage $i-1$. The output Handshake lines consist of a request line to and a grant line from stage $i+1$. | 161 |
| 1.9.6. | Extra Stage Cube network layout for the 1024-PE PASM system. | 163 |
| 1.9.7. | 1024-PE PASM system floor plan. | 165 |
| 3.3.1. | (a) 1-CU and 1-PE SIMD decoupling model. (b) Serial model. (c) N-CU and N-PE SIMD decoupling model. (d) 1-CU and N-PE SIMD decoupling model. | 286 |
| 3.4.1. | Data transfers required in PE J for the "checkerboard" data allocation. | 291 |
| 3.4.2a. | Original subimage arrays (4-by-4). | 292 |
| 3.4.2b. | Workspace arrays (6-by-6) after interprocessor data transfers. | 292 |
| 3.4.2c. | Workspace arrays (6-by-6) after copy of original subimage data. | 293 |
| 3.4.2d. | Workspace arrays (6-by-6) after edge fixup operations. | 293 |
| 3.4.2e. | Smoothed subimage (4-by-4) replaces original subimage. | 294 |
| 3.4.2f. | Workspace array (12-by-12) showing savings in number of pixels copied from original subimage. | 294 |
| 3.4.2g. | Smoothed results from operations on workspace arrays placed in result (4-by-4) array. | 296 |
| 3.4.2h. | Smoothed results from operations on original subimage arrays placed in result array (4-by-4). | 296 |
| 3.5.1. | Histogram calculation for $N = 16$ PEs, $\text{nbins} = 4$ bins. (w, \dots, z) denotes that bins w through z of the partial histogram are in the PE. | 303 |
| 3.6.1. | Results of Phase I of contour tracing for a 30-by-20 subimage. (Based on [TuA83]). | 317 |

| Figure | | Page |
|---------|--|------|
| 3.6.2. | Results of Phase II of contour tracing for a 30-by-20 subimage. (Based on [TuA83]). | 320 |
| 3.7.1. | Arcelli's templates for thinning images (Based on [Hil83]). | 328 |
| 3.7.2. | Enhanced distance algorithm; 1-valued pixels are denoted by dots or letters. Each of the four PEs has a subimage of six by six pixels. Pixels A and B can be processed immediately; C, D, and E require results calculated by other PEs. | 331 |
| 3.7.3. | Templates for identification of line ends and junctions for 8-connected imagery: (a) line end; (b) line junction; (c) middle point; (d) indeterminate point. | 337 |
| 3.7.4. | Connection of curves across subimage boundaries. G and H are the maximally distant points from line segment FA for PEs 2 and 0, respectively. | 340 |
| 3.8.1 | (a) Vector (0, 2, 2, 0) with center labeling convention. (b) Vector (0, 2, 2, 0) with upper left-hand corner labeling convention. | 349 |
| A1.6.1. | Creation of an executable program "a.out" consisting of a C-language module, a FORTRAN module, an assembly language module, and a module extracted from a library. | 480 |
| A1.6.2. | Creation of the executable simulation program <i>psst</i> . | 481 |
| A1.6.3. | Interconnection of two MC68000-based computer systems with a port. | 486 |
| A1.6.4. | Pre-defined port types in <i>PSST</i> . | 493 |
| A1.6.5. | Topology file "build.pc" for the producer-consumer simulation. | 507 |
| A1.6.6. | Assembly language program "producer.s" for the producer-consumer simulation. | 508 |
| A1.6.7. | Startup script file "start.pc" for the producer-consumer simulation. | 509 |
| A1.6.8. | Sample terminal session script for the producer-consumer simulation. | 510 |

ABSTRACT

Kuehn, James T. Ph.D., Purdue University, May 1986. The PASM Parallel Processing System: Design, Simulation, and Image Processing Applications. Major Professor: Howard Jay Siegel.

Advances in device and packaging technologies are producing incremental gains in the performance of computer systems. However, these gains are being more than offset by new applications having a need to process large data sets, a need for real-time computation, or other requirements which make them prohibitively expensive to perform on conventional computer systems. This has forced computer architects to consider parallel/distributed computer designs.

Among the variety of high-performance architectures that have been proposed or constructed, there is one common characteristic: the use of parallelism. This thesis outlines the design of a flexible parallel processing system, PASM, that can be dynamically reconfigured to meet the particular needs of a large variety of image, signal, and vector processing applications. PASM operates in two modes of parallelism: single instruction stream - multiple data stream (SIMD) mode and multiple instruction stream - multiple data stream (MIMD) mode. This allows parallel algorithms to be expressed and coded in their most natural and efficient forms.

The thesis is divided into three major parts. Part I surveys existing and proposed parallel/distributed computer architectures and describes the PASM architecture concepts. It then details work toward an implementation for PASM: conceptual studies of the interfaces between processors, controllers, interconnection network, and memory systems; simulation studies for estimating machine performance; and the evaluation and selection of components appropriate for a 30-processor PASM prototype implementation. A design proposal for a 1000-processor PASM implementation is also given.

Part II describes two parallel languages suitable for programming an SIMD/MIMD machine such as PASM: a PASM-prototype-specific assembly

language and a higher-level one based on the C programming language. Operating system issues for the PASM prototype are also studied.

Part III introduces parallel image processing algorithms that were developed and simulated to study PASM performance. These studies were used to identify desirable hardware, language, and operating system features. A section on performance measures useful for evaluating SIMD and MIMD algorithms is included.

GLOSSARY OF TERMS AND SYMBOLS

| | |
|--------|---|
| b | The branching factor in a pyramid architecture. |
| CDP | (Command Distribution Processor). One of the processors of the PASM Memory Management System that distributes work among the MSUs. |
| CPE | (Conceptual Processing Element). A Parallel-C language convention for referring to a PE in a d-dimensional space. |
| CPU | (Central Processing Unit). A device that performs arithmetic, logical, and control flow operations in a computer system. |
| CS | (Control Storage). An intelligent secondary storage device primarily used for the storage of SIMD programs in the PASM architecture. |
| CU | (Control Unit). A control processor used in SIMD architectures to fetch and broadcast instructions to the PEs. |
| DMA | (Direct Memory Access). A mode of data transfer performed under the supervision of a Direct Memory Access Controller rather than a CPU. |
| DP | (Directory Processor). One of the processors of the PASM Memory Management System that is the entry point for MSU disk service requests. |
| EEPROM | (Electrically Erase-able Programmable Read-Only Memory). A type of non-volatile computer memory that is read-only in normal use. Its contents can be changed while in-circuit using a special re-programming sequence. |
| EPROM | (Erase-able Programmable Read-Only Memory). A type of non-volatile computer memory that is read-only in normal use. Its contents can be changed by removing it from a computer circuit, erasing it with ultraviolet light, and re-programming it with specialized hardware. |
| ESC | (Extra Stage Cube). A type of fault-tolerant interconnection network based on the multistage Cube network. |

| | |
|---------|--|
| FBU | (Fetch and Broadcast Unit). A specialized component in an SIMD machine that fetches SIMD instructions and broadcasts them to active PEs. |
| H | The height (number of levels) in a pyramid architecture. |
| IOP | (Input/Output Processor). One of the processors of the PASM Memory Management System through which external data enters/exits PASM. |
| L | The level number in a pyramid processor. |
| LMC | (Logical Micro Controller). One of a set of Physical MCs chosen to participate in a fixed-sized partition. |
| LPE | (Logical Processing Element). One of a set of Physical PEs chosen to participate in a fixed-sized partition. |
| M | The number of PASM Microcontrollers assigned to a particular partition. |
| m | $m = \log_2 M$. |
| MC | (Micro Controller). A control processor used in the PASM architecture, typically used as a CU in SIMD mode and for scheduling activities in MIMD mode. |
| MC68000 | The name of a 16-bit CPU manufactured by Motorola, Inc. |
| MC68010 | The name of a 16-bit CPU manufactured by Motorola, Inc. which is an enhancement of the MC68000. |
| MC68020 | The name of a 32-bit CPU manufactured by Motorola, Inc. which is an enhancement of the MC68010. |
| MC68881 | The name of a floating point co-processor manufactured by Motorola, Inc. which is intended to be used with an MC68020. |
| MIMD | (Multiple Instruction stream - Multiple Data stream). A type of computing paradigm characterized by a set of processors each performing independent instruction streams (programs), processing different data, and communicating asynchronously. |
| MMS | (Memory Management System). One or more processors used in the PASM architecture for control of the secondary storage system. |
| MSS | (Memory Storage System). The MMS together with the set of MSUs of a PASM machine. |

| | |
|------|---|
| MSP | (Memory Scheduling Processor). One of the processors of the PASM Memory Management System that prioritizes requests for MSU disk service. |
| MSU | (Memory Storage Unit). An intelligent secondary storage device used in the PASM architecture. |
| N | The number of processors in a parallel machine that are used to perform computation (non-control) functions. |
| n | $n = \log_2 N$. |
| NIU | (Network Interface Unit). A device associated with a PE that manages inter-PE data transfers and handles the low-level interconnection network protocols. |
| I/O | (Input/Output). A generic term for non-memory data transfers, applied typically to inter-processor communication, use of mechanical storage devices, and interface with keyboards and terminals. |
| P | A processor's number, expressed in binary notation as: $P = p_{n-1}p_{n-2} \dots p_1p_0$. |
| PASM | (Partitionable SIMD/MIMD). A type of parallel processing architecture characterized by a particular arrangement of processors, memories, interconnection network, and secondary storage devices that can operate in either SIMD or MIMD mode. |
| PC | (Program Counter). A specialized register internal to a CPU that holds the address of the next program instruction to be fetched. |
| PE | (Processing Element). A basic computing element consisting of a processor and its own local memory. |
| PMC | (Physical Micro Controller). An MC (compare with LMC and VMC). |
| PPE | (Physical Processing Element). A PE (compare with LPE, VPE, and CPE). |
| PROM | (Programmable Read-Only Memory). A type of read-only non-volatile computer memory. Its contents, once written, cannot be changed. |
| Q | The number of Microcontrollers in a PASM computer. |
| q | $q = \log_2 Q$. |
| RAM | (Random Access Memory). Volatile read/write memory. |

| | |
|------|---|
| ROM | (Read-Only Memory). A generic term for non-volatile read-only memories such as PROMs, EPROMS, and EEPROMS. |
| SCU | (System Control Unit). A processor in the PASM architecture dedicated to system control and scheduling functions. |
| SIMD | (Single Instruction stream - Multiple Data stream). A type of computing paradigm characterized by a set of processors each performing the same instruction stream (program) synchronously, processing different data, and communicating synchronously. Synchronization and control is enforced by the CU. |
| VMC | (Virtual Micro Controller). An MC emulated by a PMC. When the number of PMCs meets or exceeds the number of VMCs required by a program, optimum performance is attained. |
| VME | (Versa-Module European). A particular bus standard commonly used by Motorola 68000-based computer systems. |
| VPE | (Virtual Processing Element). A PE emulated by a PPE. When the number of PPEs meets or exceeds the number of VPEs required by a program, optimum performance is attained. |

THESIS MATERIAL PREVIOUSLY PUBLISHED BY THE AUTHOR

The author has previously published portions of the work presented in this thesis in the open literature. Titles of these publications and the corresponding thesis sections in which the material appears are listed below.

- [1] J. T. Kuehn and H. J. Siegel, "Simulation studies of PASM in SIMD mode," *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, November 1981, pp. 43-50. (Material related to Sections I.5 and III.4.)
- [2] J. T. Kuehn, H. J. Siegel, and P. D. Hallenbeck, "Design and simulation of an MC68000-based multimicroprocessor system," *1982 International Conference on Parallel Processing*, August 1982, pp. 353-362. (Material related to Sections I.5 and III.4.)
- [3] J. T. Kuehn, H. J. Siegel, and M. J. Grosz, "A distributed memory management system for PASM," *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, October 1983, pp. 101-108. (Material related to Sections I.5 and II.5.)
- [4] J. T. Kuehn and H. J. Siegel, "Simulation studies of a parallel histogramming algorithm for PASM," *Seventh International Conference on Pattern Recognition*, July 1984, pp. 646-649. (Material related to Section III.5.)
- [5] H. J. Siegel, T. Schwederski, N. J. Davis IV, and J. T. Kuehn, "PASM: a reconfigurable parallel system for image processing," *Workshop on Algorithm-guided Parallel Architectures for Automatic Target Recognition*, July 1984, pp. 263-291. (Also appears in the ACM SIGARCH newsletter: *Computer Architecture News*, Vol. 12, No. 4, September 1984, pp. 7-19). (Material related to Sections I.5 and I.6.)
- [6] D. G. Meyer, H. J. Siegel, T. Schwederski, N. J. Davis IV, and J. T. Kuehn, "The PASM parallel system prototype," *IEEE Computer Society Spring Compcon 85*, February 1985, pp. 429-434. (Material related to Sections I.5 and I.6.)
- [7] J. T. Kuehn, J. A. Fessler, and H. J. Siegel, "Parallel image thinning and vectorization on PASM," *1985 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, June 1985, pp. 368-374. (Material related to Section III.8.)
- [8] J. T. Kuehn and H. J. Siegel, "Extensions to the C programming language for SIMD/MIMD parallelism," *1985 International Conference on Parallel Processing*, August 1985, pp. 232-235. (Material related to Section II.4.)

- [9] H. J. Siegel and J. T. Kuehn, "PASM: a partitionable SIMD/MIMD system for parallel image processing research," in *Algorithmically Specialized Parallel Computers*, L. Snyder, L. H. Jamieson, D. B. Gannon, and H. J. Siegel, eds., Academic Press, New York, NY, 1985, pp. 69-78. (Material related to Sections I.5 and I.6.)
- [10] J. T. Kuehn, H. J. Siegel, D. L. Tuomenoksa, and G. B. Adams III, "The use and design of PASM," in *Integrated Technology for Parallel Image Processing*, S. Levialdi, ed., Academic Press, San Diego, CA, 1985, pp. 133-152. (Material related to Sections I.5 and III.8.)
- [11] J. T. Kuehn, T. Schwederski, and H. J. Siegel, "Design of a 1024-processor PASM system," *First International Conference on Supercomputing Systems*, December 1985, pp. 603-612. (Material related to Sections I.7 and I.8.)
- [12] J. T. Kuehn and H. J. Siegel, "Simulation based performance measures for SIMD/MIMD processing," in *Computing Structures and Image Processing*, K. Preston, Jr., L. Uhr, M. J. B. Duff, and S. Levialdi, eds., Academic Press, San Diego, CA, to appear, 1986. (Material related to Sections III.3, III.4, III.5, and III.8.)
- [13] J. T. Kuehn and H. J. Siegel, "Multifunction processing with PASM," *Intermediate-Level Image Processing*, M. J. B. Duff, ed., Academic Press, London, England, to appear, 1986. (Material related to Sections I.5 and I.6.)

PART I**PASM AND THE PASM PROTOTYPE**

CHAPTER 1

INTRODUCTION

Advances in device and packaging technologies are producing incremental gains in the performance of computer systems. However, these gains are being more than offset by new applications having a need to process large data sets, a need for real-time computation, or other requirements which make them prohibitively expensive to perform on conventional computer systems. This has forced computer architects to consider parallel/distributed computer designs.

In Part I of this thesis, the strengths and weaknesses of a variety of existing and proposed parallel computer designs are examined. These characteristics are used to motivate and support the design decisions made for a particular parallel computer architecture, PASM. It is PASM that is the primary focus of the work in this thesis.

The remainder of Part I is organized as follows. Chapter 2 introduces common parallel computation models and interconnection structures. Existing and proposed parallel processing system architectures are summarized in Chapter 3. A particular parallel machine, PASM, is overviewed in Chapter 4. Chapter 5 gives a brief history of the PASM development. A detailed description of the design studies performed for PASM is given in Chapter 6. These studies helped to influence the design and implementation of the 16-processor PASM prototype discussed in Chapter 7. Chapter 8 reviews some of the lessons learned from the prototype development. A design proposal for a 1024-processor PASM system is given in Chapter 9. Part I is summarized in Chapter 10. Much of the material in Part I has been previously published by the author as a part of [SiK81, SiK82, KuS83, KuS81, KuS82, KuS84, SiS84, SiK85, MeS85, KuS85, KuS86a].

CHAPTER 2

PARALLEL COMPUTATION MODELS

1.2.1 SIMD Model

One type of parallel processing system is the single instruction stream - multiple data stream (SIMD) machine [Fly66]. An *SIMD machine* typically consists of a *Control Unit (CU)*, an interconnection network, N processors, and N memories. The CU broadcasts instructions to the processors and all enabled processors execute the same instruction or use the interconnection network at the same time. This is the single instruction stream. While an instruction is being executed, each of the N processors is connected to a different memory (either by a fixed connection or a connection provided by an interconnection network) and thus each of the N processors operates on different data. These make up the multiple data streams. The processors need not have control capabilities (e.g., program counter, branch instructions) because program control is provided by the CU.

There are two basic ways in which the processors and memories of SIMD machines are arranged. One way, shown in Figure 1.2.1, is to form processor/memory pairs called *Processing Elements (PEs)* in which a processor can directly access only the data in its own *local* memory. The PEs are numbered from 0 to N-1 and each PE knows its number (address). The configuration shown in Figure 1.2.1 is known as the PE-to-PE model [SiS81b] because the PEs communicate among themselves using the interconnection network. Local memory references are relatively fast; however, transfers of data from PE to PE are less efficient due to interconnection network propagation delays and the memory fetching and storing that occurs for each item transferred.

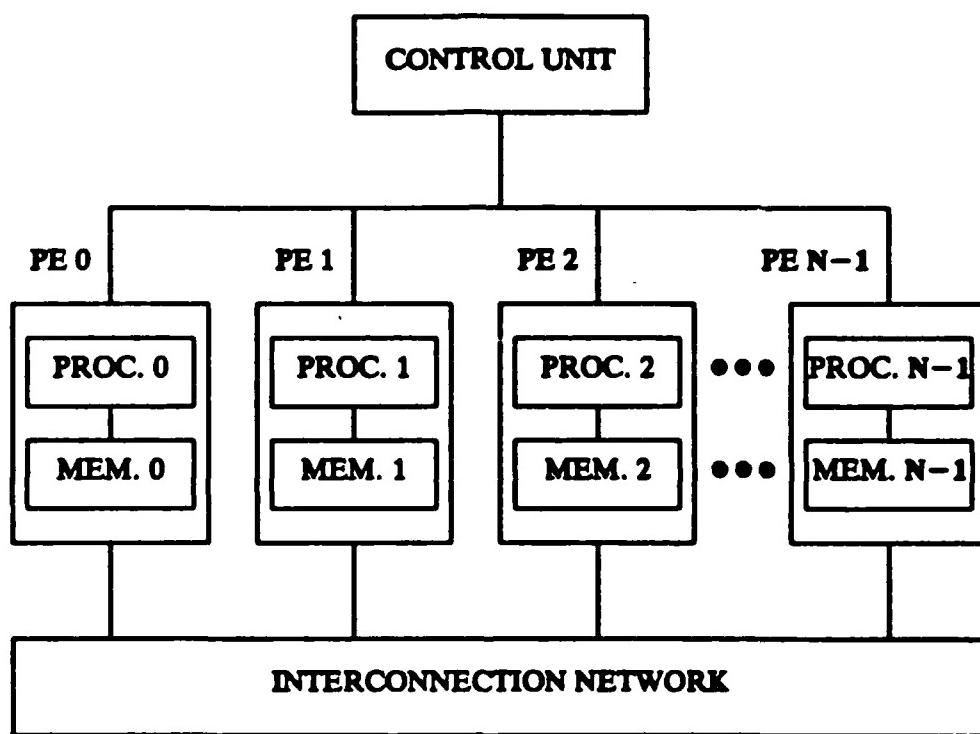


Figure 1.2.1. SIMD/MIMD machine model (PE-to-PE configuration).

An alternative configuration is the P-to-M (processor-to-memory) model [SiS81b] of Figure 1.2.2 in which the interconnection network is used to connect the processors to the memory modules. Here, all of the enabled processors simultaneously send the global address of a data item to be fetched through the interconnection network to a memory. The addressed memory returns the requested item along the reverse path. So long as the N processors generate addresses that refer to different memories, the SIMD machine operates correctly. The advantage of this approach is that data can be "transferred" from one processor to another simply by having the processors generate different addresses. A disadvantage is that all memory references must go through the interconnection network which is usually a slower path than that to a local memory. A detailed analysis of the tradeoffs between the two configurations appears in [SiS81b].

A subclass of SIMD machines is the associative processor [ThW75, YaF77]. Each of the N processors in an associative processor has logic that allows it to match a bit of its data word against a bit of a template word that is managed by the CU. Enabled processors that find matches respond affirmatively to the CU. The machine is classified as bit-associative if only one bit is matched per processor and word-associative if a number of bits may be matched simultaneously in each processor. If the associative processor is configured with processor/memory pairs, the interconnection network need not be implemented.

1.2.2 MIMD Model

Another type of parallel processing system is a multiple instruction stream - multiple data stream (MIMD) machine [Fly66]. An *MIMD machine* typically consists of N general-purpose processors, N memories, and an interconnection network. The processors follow independent (multiple) instruction streams and process multiple data streams. MIMD machines are often referred to as multiprocessors.

As with SIMD architectures, there are a number of arrangements of the processors and memories in an MIMD machine. In the PE-to-PE model, processors and memories are paired to form PEs and the PEs use the interconnection network to communicate by message-passing. Since the PEs are operating

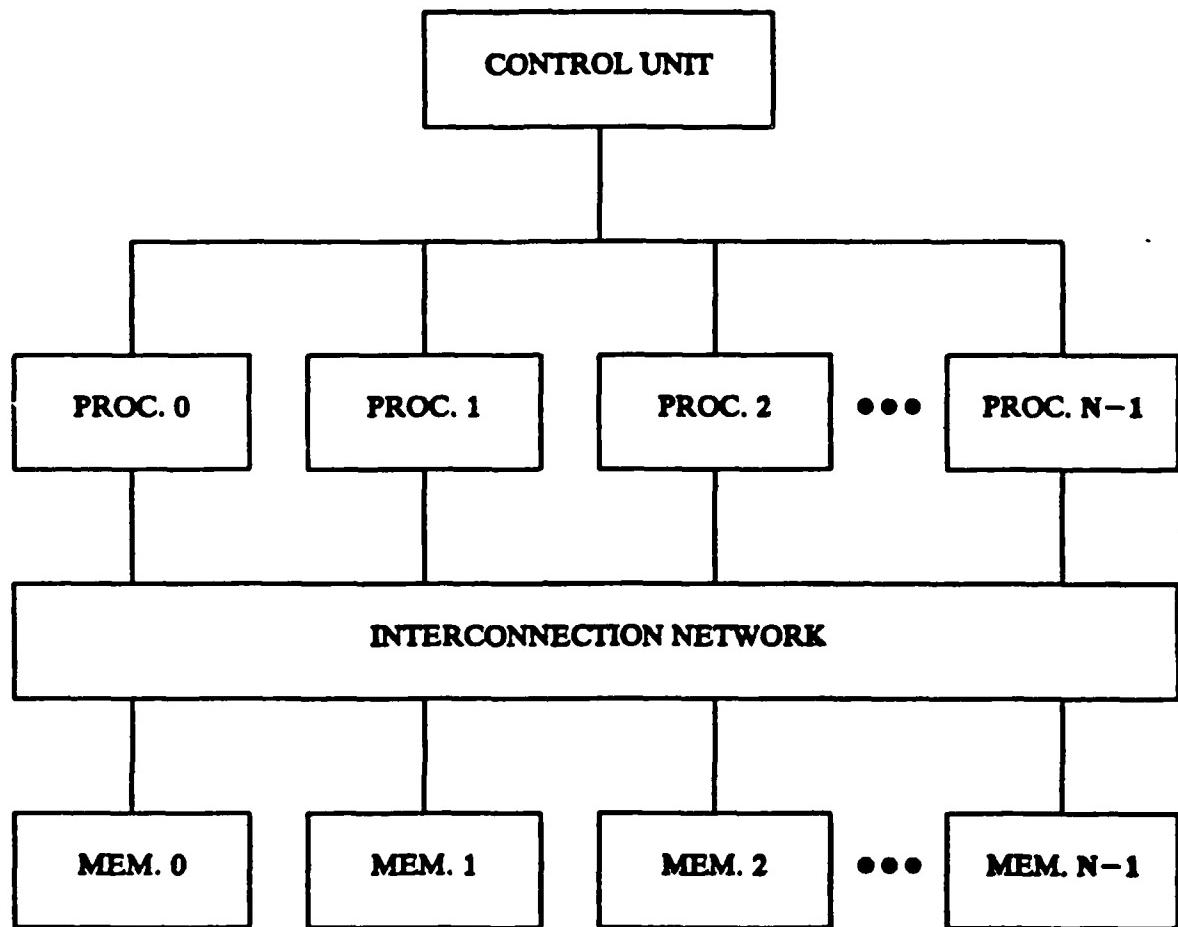


Figure 1.2.2. SIMD/MIMD machine model (P-to-M configuration).

asynchronously, there is no requirement that the interconnection network support simultaneous communication between all N PEs as was the case for SIMD mode. Such machines are often characterized as *loosely-coupled*, *distributed*, or *message-passing* systems.

In the P-to-M model, an interconnection network connects the processors to the memories, allowing memory to be shared among the processors. The number of memories need not be the same as the number of processors. Such a machine is often characterized as a *tightly-coupled* or *shared-memory* system.

1.2.3 MSIMD and MMIMD Models

An *MSIMD (multiple-SIMD) system* [Nut77a] is a parallel processing system with multiple CUs, N processors and memories, and a partitionable interconnection network that allows it to be structured as one or more independent SIMD machines. An independent sub-machine will be referred to as a *partition*. The connections between a CU and a set of PEs may be fixed, limiting the numbers and sizes of partitions that can be formed, or may be flexible. The partitionability of the interconnection network also places constraints on the number and sizes of machine partitions.

An *MMIMD (multiple-MIMD) system* is a parallel processing system with N processors and memories and a partitionable interconnection network that allows it to be structured as one or more independent MIMD machines. As with MSIMD architectures, the partitionability of the interconnection network places constraints on the number and sizes of machine partitions.

1.2.4 Pipeline Model

Pipelining is another form of parallelism in which each processor follows an independent instruction stream but data flows from the output of one processor to the input of the next [RaL77]. This type of parallelism is sometimes considered to be multiple instruction stream - single data stream (MISD) in nature since a single stream of data is operated upon by multiple processors. The "pipeline" structures are usually highly regular for efficient semiconductor integration. Typically, the connections between the processors are fixed and

the processors themselves are uni-functional. This limits the pipeline's use to specific, pre-wired algorithms. Common structures include simple 1-D arrays, 2-D square or 2-D hexagonal arrays, trees, and triangular arrays of processors. These regular structures are sometimes referred to as "systolic" architectures [Kun82].

Another example of a parallel architecture in which the data moves from processor to processor to be operated upon is a *dataflow machine* [Rum77, TrB82]. Unlike the simple processor arrays, dataflow machines have multifunction processors and a flexible interconnection structure which gives them general-purpose computing ability.

1.2.5 Hybrids

Because parallelism has been applied to almost every aspect of computer design, real architectures are likely to be hybrids whose constituent elements are based on a variety of architecture models. Hybrid architectures result from a realization of the limitations of using of any single architecture model alone; they are a selective application of various architectures dictated by cost and performance requirements. For example, suppose the PEs of an MIMD machine share a special-purpose arithmetic pipeline unit. Certainly, the arithmetic operations provided by the pipeline could have been performed in the PEs themselves, but this approach would have sacrificed performance. On the other hand, the pipeline could have been replicated and integrated with each of the N PEs; however, this might not have been cost-effective.

1.2.6 Performance Characteristics

SIMD vs. MIMD

SIMD parallelism has shown tremendous performance for structured tasks with large homogeneous data sets. Many examples of highly efficient SIMD algorithms come from the image processing domain: clipping, smoothing [SiS81b, KuS82], histograming [SiS81c, KuS84], and 2-D correlation [SiS82a]. This is because these algorithms operate on the large and homogeneous pixel-level representation of an image. The SIMD machine processors can be used for "local" processing of the pixels of the image in parallel because all pixels can be treated identically. Matrix arithmetic for such tasks as statistical pattern recognition can also be done efficiently in SIMD mode.

SIMD computations are less effective when the data is non-homogeneous or unevenly distributed among the processors. Because the SIMD processors are synchronized, uneven distribution of data causes some processors to be idled while others work. Non-homogeneous data implies "special-case" processing: since the code for only one "case" at a time can be executed, there are potentially many processors idled at any given time.

The processors of SIMD machines act as slave units: they depend entirely on their CU to feed them instructions and to enable/disable them. The processors generally have a non-standard instruction set (due to the lack of a program counter, stack pointer, and control flow instructions). While this simplifies the processor itself, possibly yielding higher performance due to reductions in circuit complexity and increases in circuit density, any machine that operates strictly in SIMD mode suffers from some fundamental limitations. The most obvious limitation is the type of tasks that may be run on a strict SIMD machine. Because there is a single instruction stream, PEs must either be performing the current instruction or be disabled. For example, as a result of executing the conditional branch statement:

```
if A > B then C := A else C := B
```

where A, B, and C are vectors (an element of which resides in each PE), some PEs find their A value to be greater than their B value while other PEs do not. Therefore, some PEs will need to execute the "then" part while the others will

need to execute the "else" part. Since the CU can broadcast at most one instruction at a time, it first enables the PEs for which the condition is true and broadcasts the instructions for the "then" part. Later, it enables the PEs for which the condition is false and broadcasts the "else" part. The situation becomes even worse when conditionals are nested or other looping constructs are used. In the worst case, an SIMD algorithm which has nested conditionals may be executed no better than serially.

These limitations of SIMD processing are erased by allowing multiple independent instructions streams, as does MIMD operation. Again using the image processing problem domain as an example, MIMD "pattern recognition" algorithms may be used to find two or more distinct objects in an image: each object is assigned a processor or set of processors to search for it. MIMD parallelism has been more successful with the image processing tasks that deal with higher-level non-homogeneous image representations such as lines, contours, and regions. These structures cannot generally be treated uniformly by SIMD image processing algorithms. For example, more complicated areas of an image generally require more data to represent them. Unfortunately, these higher-level data representations cannot be so easily be distributed among processors to equalize the work load. Often they represent characteristics of an image that are more global in nature and thus resist being distributed among multiple processors.

The flexibility of MIMD architectures comes at a price however. Each MIMD processor must be a complete general-purpose computation unit; by comparison, SIMD processors do not require control functions. The explicit synchronization between SIMD processors also has its advantages: control over decision-making is centralized, synchronous interconnection networks are more easily controlled, and there is no software overhead for the synchronization. Thus a potentially useful hybrid architecture is one that can act either as an SIMD machine or MIMD machine and can dynamically switch between modes. Such a system allows algorithms to be coded and executed in their most natural and efficient mode of parallelism. Also, data to be processed using both SIMD and MIMD algorithms need not be moved from machine to machine since it can be processed using the same PEs.

MSIMD vs. SIMD

The possible advantages of a system capable of operating as a multiple-SIMD machine over an SIMD machine with a similar number of processors include the following [SiS81b]:

- (1) *Partitioning for Utilization*: If a task requires only $N/2$ of N available processors, the other $N/2$ can be used for another task.
- (2) *Fault Detection*: For situations where high reliability is needed, three partitions can run the same program on the same data and compare results.
- (3) *Fault Tolerance*: If a single processor fails, only those logical SIMD machine partitions which must include the failed processor need to be disabled. The rest of the system can continue to function.
- (4) *Multiple Simultaneous Users*: Since there can be multiple independent SIMD machine partitions, there can be multiple simultaneous users of the system, each executing a different SIMD program.
- (5) *Partitioning for Program Development*: If the partition size is flexible, rather than trying to debug an SIMD program on, for example, 1024 processors, it can be debugged on a smaller size SIMD machine.
- (6) *Subtask Parallelism*: Two independent SIMD subtasks that are part of the same job can be executed in parallel, sharing results if necessary.

Similar advantages apply when comparing an MMIMD architecture to an MIMD architecture.

1.2.7 Interconnection Structures

Network Characteristics

Interconnection networks are critical elements in the design of parallel and distributed computer systems. The choice of an interconnection network depends on many factors; these include the bandwidth required, the physical distance between the processors to be interconnected, the number of processors to be connected, the fault-tolerant capabilities required, and the allowable cost. In this subsection, examples of interconnection structures most often used to connect the processors of SIMD and MIMD machines ("intra-machine"

networks) are given. Other types of networks such as "local-area" and "long-haul" networks may have some of the characteristic topologies indicated below, but have radically different implementation and control schemes. Aspects of these "inter-machine" networks are treated in [Tan81a]. The following discussion of network taxonomies is loosely based on the one presented in [AnJ75].

The *cost* and complexity of an interconnection network is often estimated by a count of the number of switches and (bidirectional) links required to implement it for N processors. The *flexibility* of a network is related to the number of *interconnection functions* (mappings on the set of processor addresses) [Sie79] it can perform. The *performance* of the network is determined by how many simultaneous connections between processors can be established and the best, average, and worst-case path lengths between a source and destination processor. For example, if a network has " $P+1$ " and " $P-1$ " interconnection functions, it means that each processor " P " has direct links to the processors numbered $P+1$ and $P-1$. If a processor wishes to communicate with a processor to which it is not directly connected, for example, one that is $+5$ away, five applications of the " $+1$ " interconnection function would be required. Thus the performance of a network is a function of the number of intervening links, switches, or processors the average message must pass through on its way from a source to a destination processor. The performance will be described in terms of the maximum number of steps (applications of interconnection functions) required to move data between a pair of processors in a network that is correctly functioning.

Networks with *centralized control* are often limited in their ability to perform simultaneous interconnection functions, i.e., processors must all communicate using either the $+1$ or the -1 function, but not both, at any given time. Networks with *distributed control* may be able to perform multiple interconnection functions simultaneously, with the caveat that no "conflicts" occur. Conflicts are a result of two messages requiring the same network link simultaneously. Some networks are capable of resolving the conflict by arbitrating the access to the network; others are not.

Fault-tolerance is a measure of how well the network performs when one or more faults occur. A network is said to have *n-fault-tolerance* if it can continue to allow all pairs of non-faulty processors to communicate in the presence

of any set of n faults in links, switches, or intervening processors. The network is considered fault-tolerant even if there is performance degradation; e.g., a message must take a longer path to avoid the faults. For a network to be n -fault-tolerant, there must be at least $n+1$ different paths between any source-destination processor pair.

The *partitionability* of an interconnection network is its ability to be subdivided such that each of its subnetworks has the same connection properties (interconnection functions) as the original [Sie80]. Each group of one or more processors and the subnetwork they share is called a partition. Such a subdivision must be performed through isolation of the partitions; no connections between processors not originally connected may be established during the partitioning. Networks that have centralized control schemes cannot be partitioned because each subnetwork requires independent control. Although networks that can be physically partitioned are most suitable for use in MSIMD or MMIMD systems, software partitioning enforced by a machine's operating system can also be used.

Network Types

A simple and quite common interconnection structure is the *bus*, shown in Figure 1.2.3. Its advantage is its low cost of N links and N switches for connecting N processors. However, its performance is limited in that only one processor can be the "bus master" and use the bus at a time; thus it allows only one simultaneous connection. Another disadvantage is that the bus structure cannot be expanded readily to more than a dozen or so processors due to bus contention. Also buses are generally not extended over physical distances greater than about ten feet because beyond this length, signal propagation delays begin to adversely affect the performance. The bus is not fault-tolerant since there are no redundant signal paths and if a processor fails while it is the bus master, the bus arbitration protocol can be disrupted, leading to complete failure. In theory, buses are partitionable if adjacent processors are grouped and isolation is provided between the groups. In practice however, a bus' rather stringent electrical and termination characteristics make partitioning it impractical.

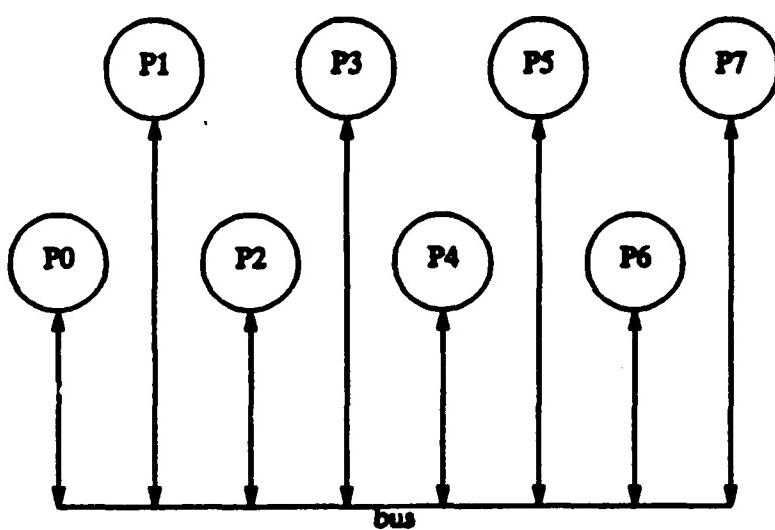


Figure 1.2.3. Bus network for $N=8$.

Another inexpensive way of interconnecting processors is via an *arbitrated shared memory* shown in Figure 1.2.4. The low cost of N links is an advantage, but performance is limited in that only one processor can access the memory at once. Memory contention prevents this structure from being practical for more than a few processors. A *multiport memory* allows simultaneous reads and writes by all processors connected to it (except for simultaneous writes to the same location). This gives higher performance, but the memory is considerably more costly. If a processor-memory link becomes faulty, only that processor is effectively disconnected; however, a faulty memory is catastrophic. Physical partitioning of shared memories is generally not possible; however, logic external to the memory can be added to logically partition the memory space and to restrict access to each partition.

A *complete interconnection* (Figure 1.2.5) provides a direct link between every pair of processors, resulting in a cost of $N(N-1)$ links. Therefore, implementation of such a network is practical only for very small N . Of course, the performance is the highest possible (one step in the worst case) since N simultaneous connections are possible and there are no intervening processors or switches. Fault tolerance is also high ($N-2$ -fault-tolerant) because there are so many redundant paths. A crossbar network is similar to a complete interconnection in that it provides direct links between every pair of processors. Essentially, it consists of N^2 switches that can be set to tie the buses of any combination of processors together. Partitioning is accomplished by disallowing communication over certain links or through certain switches. The "complete" and "crossbar" interconnections are the only ones in which the partitioning is unrestricted.

A *unidirectional ring* connection (Figure 1.2.6) is a low-cost (N links) interconnection that allows simultaneous transfers between adjacent processors using the interconnection function:

$$\text{shift}_{+1}(P) = P + 1 \text{ (modulo } N\text{)}$$

However, loss of any single link or processor effectively disconnects the ring and prevents some pairs of processors from communicating. A *bidirectional ring* (Figure 1.2.7) implements two interconnection functions:

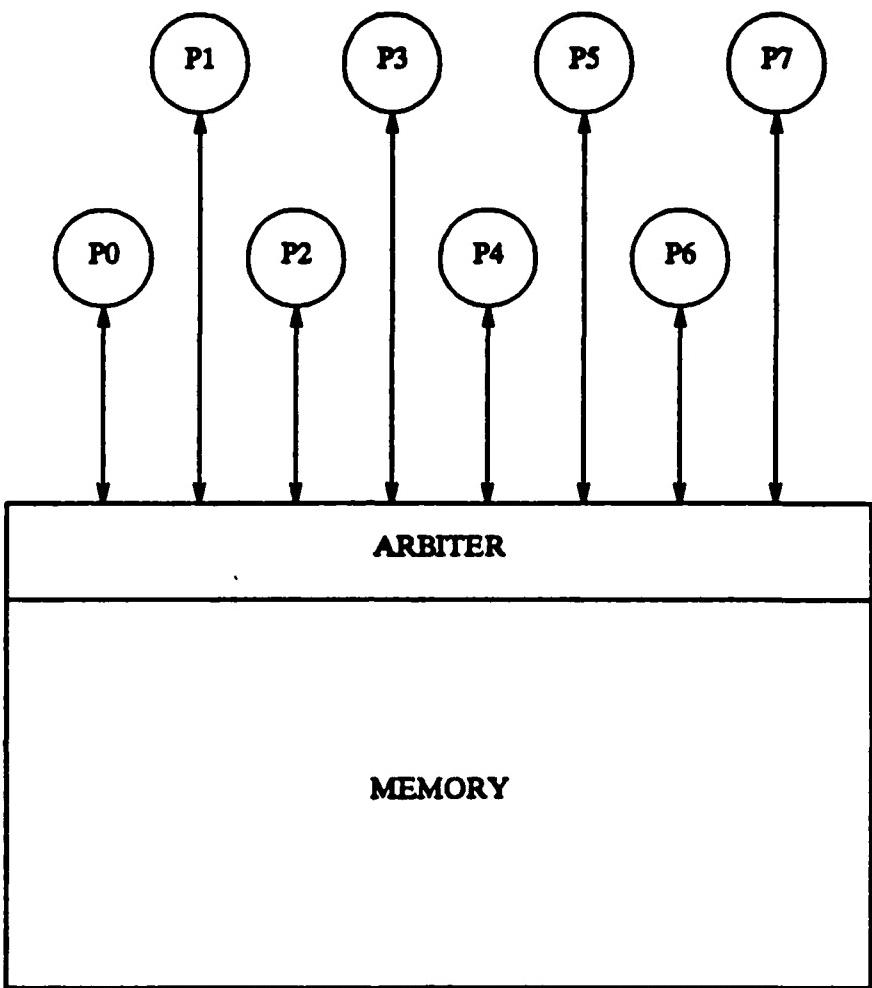


Figure 1.2.4. Shared memory connection for $N=8$.

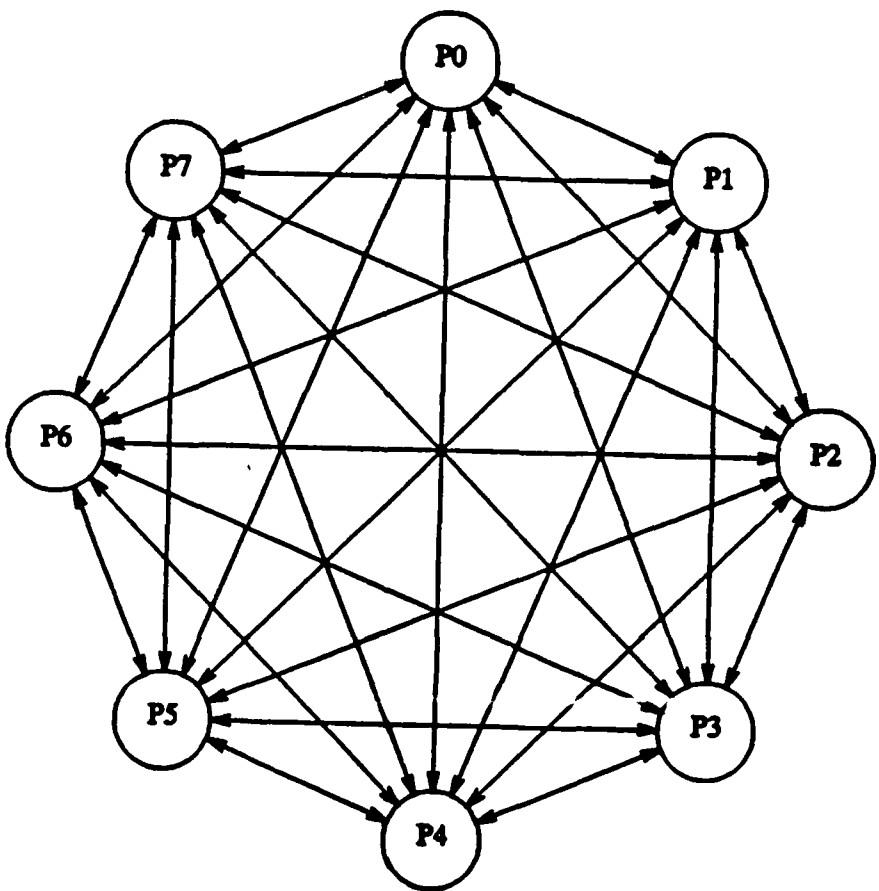


Figure 1.2.5. Complete interconnection network for $N=8$.

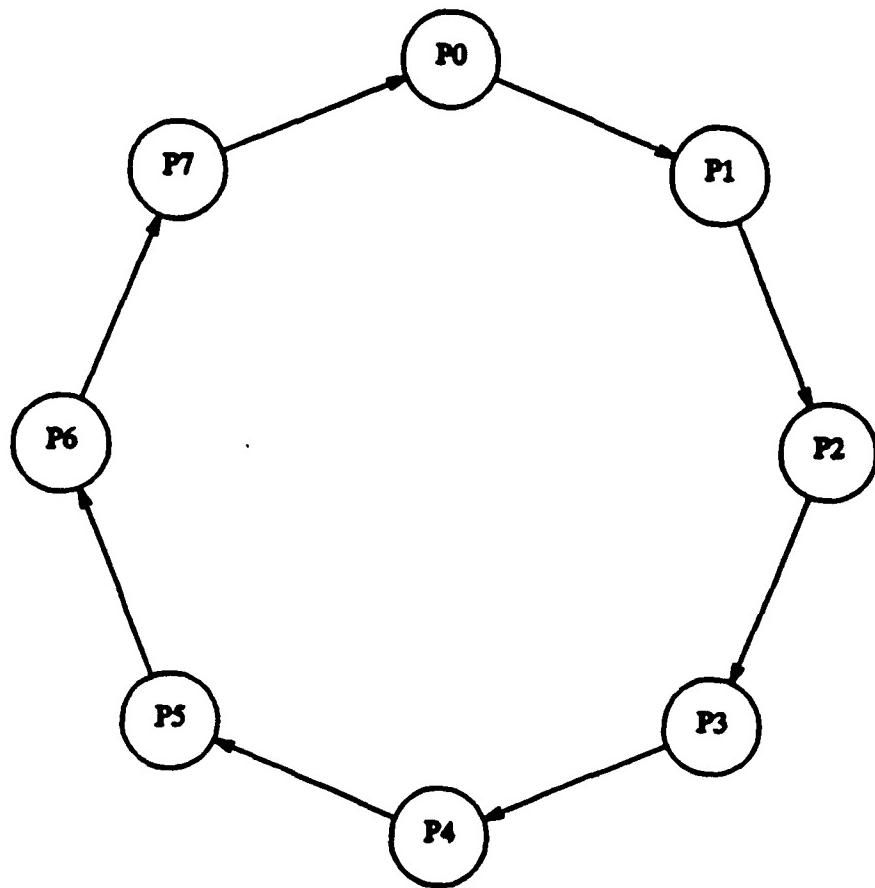


Figure 1.2.6. Unidirectional ring network for $N=8$.

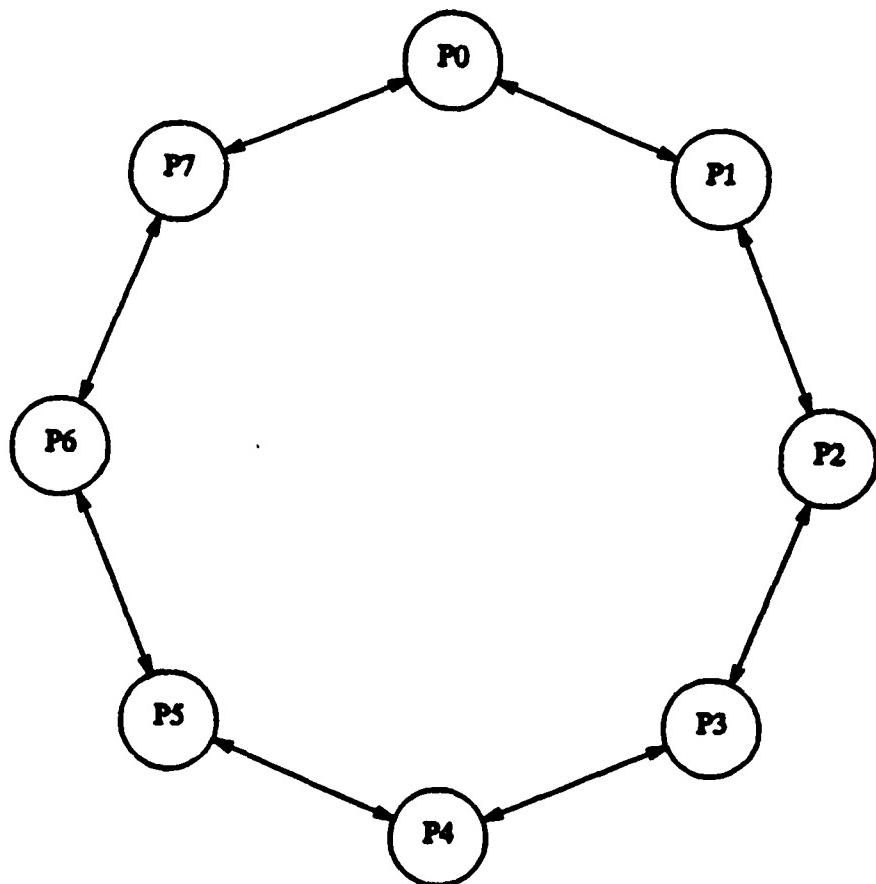


Figure 1.2.7. Bidirectional ring network for $N=8$.

$$\text{shift}_{+1}(P) = P + 1 \text{ (modulo } N)$$

$$\text{shift}_{-1}(P) = P - 1 \text{ (modulo } N).$$

It tolerates single link or processor failures (1-fault-tolerance). Performance is limited for both types of rings because messages sent between non-adjacent processors must be examined by each of the intervening processors as they pass. The maximum number of steps is $N-1$ for the unidirectional ring and $(N/2)-1$ for the bidirectional ring. Therefore, rings are not typically used to connect large numbers of processors. Rings are not partitionable since any partitioning breaks the ring.

In systems where the processors are logically arranged in a square array, a *nearest-neighbor* connection is often used. One type is the *Illiac* network (Figure 1.2.8) which connects each processor with its four nearest neighbors. Hence it performs the interconnection functions:

$$\text{Illiac}_{+1}(P) = P + 1 \text{ (modulo } N)$$

$$\text{Illiac}_{-1}(P) = P - 1 \text{ (modulo } N)$$

$$\text{Illiac}_{+\sqrt{N}}(P) = P + \sqrt{N} \text{ (modulo } N)$$

$$\text{Illiac}_{-\sqrt{N}}(P) = P - \sqrt{N} \text{ (modulo } N).$$

The Illiac network has a cost of $2N$ links and is 3-fault-tolerant. Up to \sqrt{N} steps are required for some messages. Illiac networks are not partitionable.

A connection pattern related to the Illiac network connects each processor with its eight nearest neighbors (Figure 1.2.9). This network has a cost of $4N$ links and is 7-fault-tolerant. Performance is better than the Illiac network since at most $\sqrt{N}/2$ steps are required for any message. The eight-nearest-neighbor network is not partitionable.

A *pyramid* interconnection scheme has been proposed for some computer vision applications [Tan81b, Uhr81, Uhr83] (Figure 1.2.10). A pyramid of height H , where level 0 is the apex and level $H-1$ is the base, and having a "branching factor" of b has b^L processors in level L , $0 \leq L < H$. Each processor in level L is connected to b processors "below it" in level $L+1$. When b is two, the interconnection conceptually forms a "binary tree" of processors. Larger b values (typically $b=4$) result in a "pyramid" of processors. Also, it is usual when $b=4$ to have all of the processors within a given level connected with a four-nearest-neighbor (Illiac) network. Ignoring any intra-level network

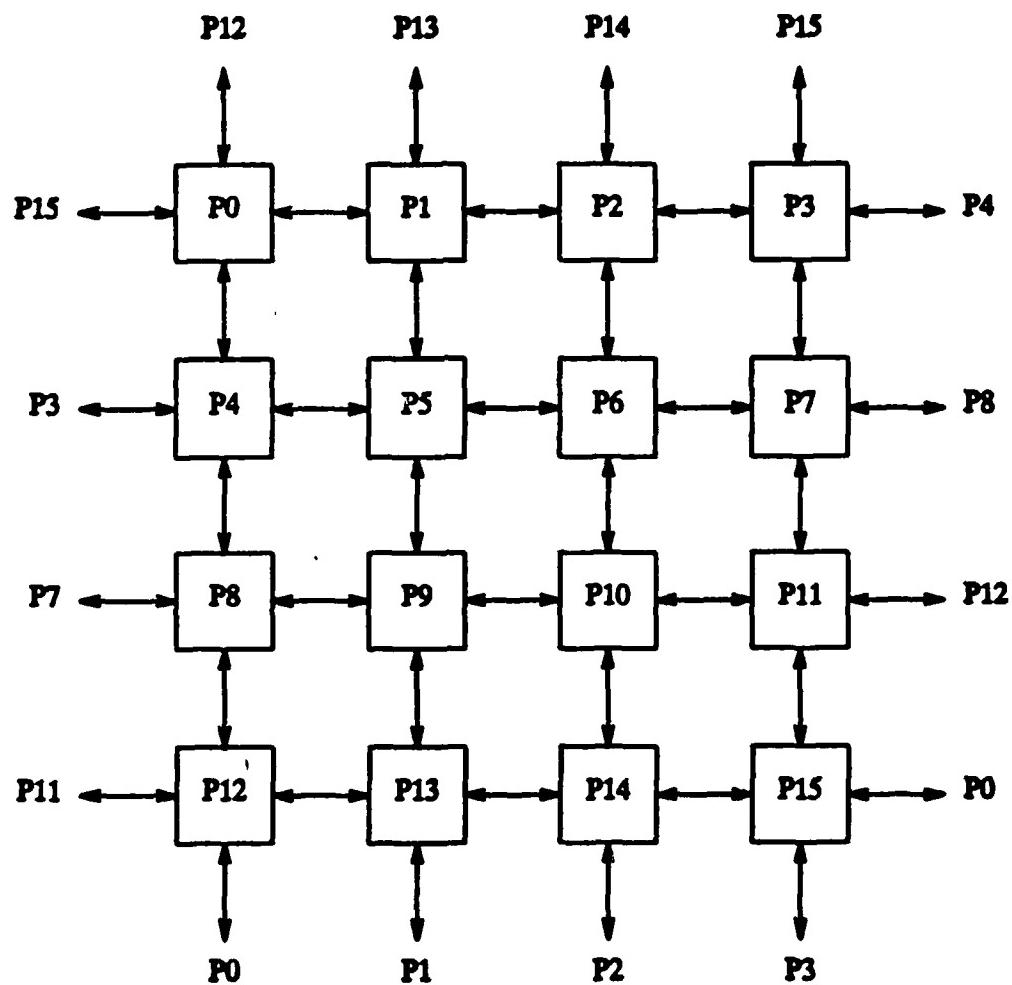


Figure 1.2.8. Four-nearest-neighbor (Illiac) connection for $N=16$.

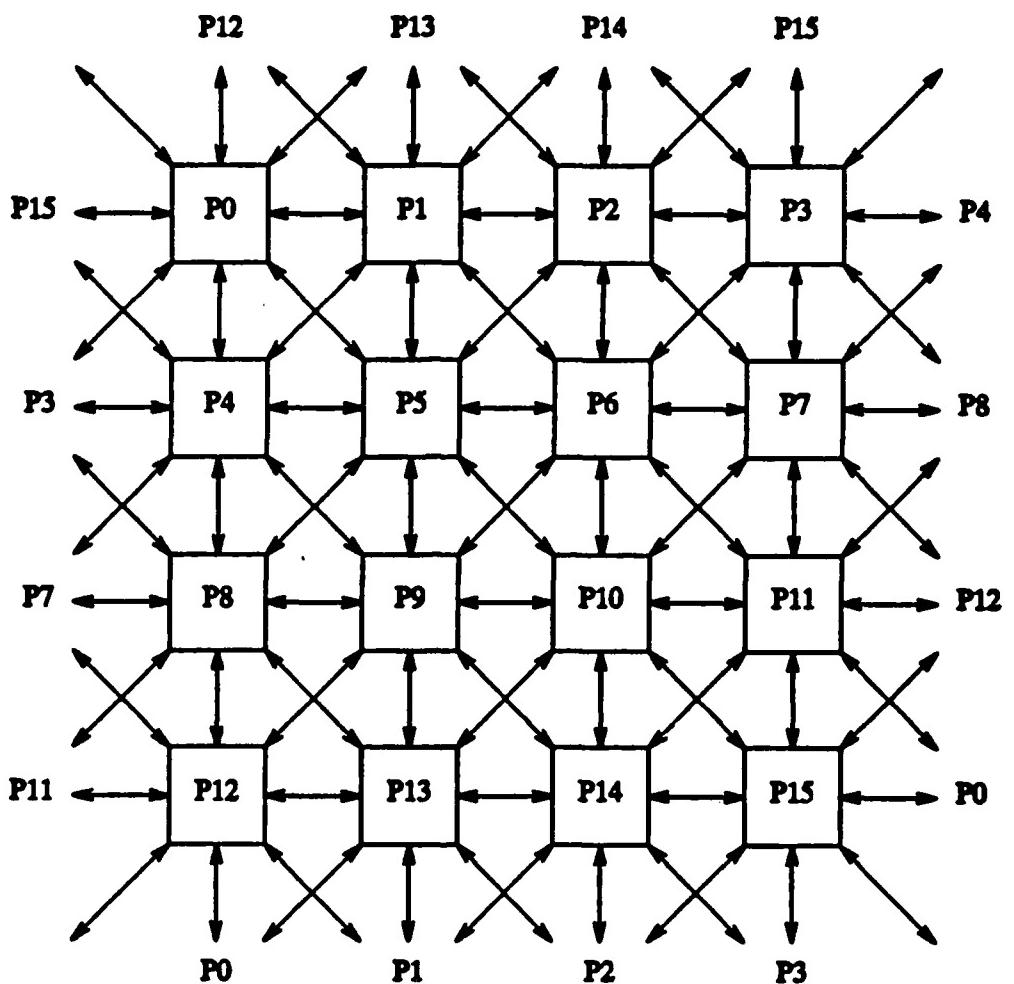


Figure 1.2.9. Eight-nearest-neighbor connection for $N=16$.

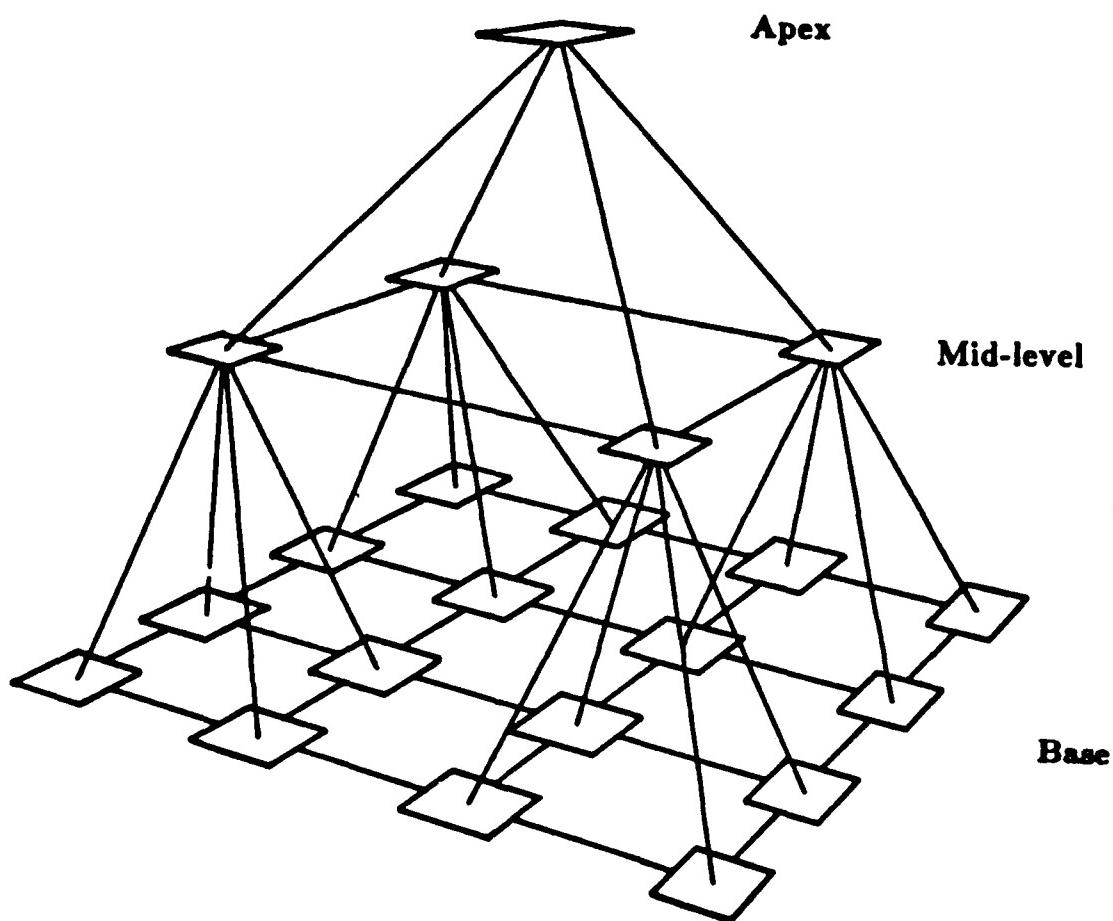


Figure 1.2.10. Pyramid connection for $H=3$, $b=4$.

connections, the number of links needed is given by

$$\sum_{i=1}^{H-1} b^i$$

and the maximum number of network steps is $2(H-1)$. Without intra-level connections, there is no fault tolerance. Also, there is the likelihood with this structure that the apex processor will become a communications bottleneck. Pyramids can be partitioned by severing all of the links between two adjacent levels. This forms $b+1$ partitions, b "below" the point that was cut and one above, each having the characteristics of a pyramid.

A *perfect shuffle* network [Sto71, LaS76] (Figure 1.2.11) can be used to interconnect a system of $2^n=N$ processors with $3N/2$ bidirectional links. The *shuffle* interconnection function maps a processor address of the form $P=p_{n-1}p_{n-2}\dots p_1p_0$ to the address $p_{n-2}\dots p_1\bar{p}_0p_{n-1}$. The *exchange* interconnection function maps P to the address $p_{n-1}p_{n-2}\dots p_1\bar{p}_0$, where \bar{p}_0 indicates the complement of p_0 . As with the ring, N simultaneous connections can be made, but performance is higher with the shuffle-exchange network since messages require a maximum of $2n-1$ steps on the way to their final destination. This is at the price of fault-tolerance however, since the loss of a link may prevent some pairs of processors from communicating. The perfect shuffle network cannot be partitioned.

Another type of interconnection network is the *Plus-Minus 2ⁱ* (PM2I) network (Figure 1.2.12). Its interconnection functions are described by:

$$PM2I_{+i}(P) = P + 2^i \text{ (modulo } N), 0 \leq i < n$$

$$PM2I_{-i}(P) = P - 2^i \text{ (modulo } N), 0 \leq i < n$$

for a total of Nn links. Performance is better than the perfect shuffle network since a maximum of n steps are required by any message. This network is $2n-2$ -fault-tolerant. The PM2I network is partitioned into two sub-networks by disallowing communication that uses the $PM2I_0$ function pair; i.e., $P+2^0(\text{modulo } N)$ and $P-2^0(\text{modulo } N)$. A partition consists of those processors whose addresses match in the 0'th bit; processors whose addresses differ in the 0'th bit are prevented from communicating. Further subdivision is performed by disallowing communication using the $PM2I_1$ function pairs and continuing, in order, until the $PM2I_{n-1}$ function pair is disallowed.

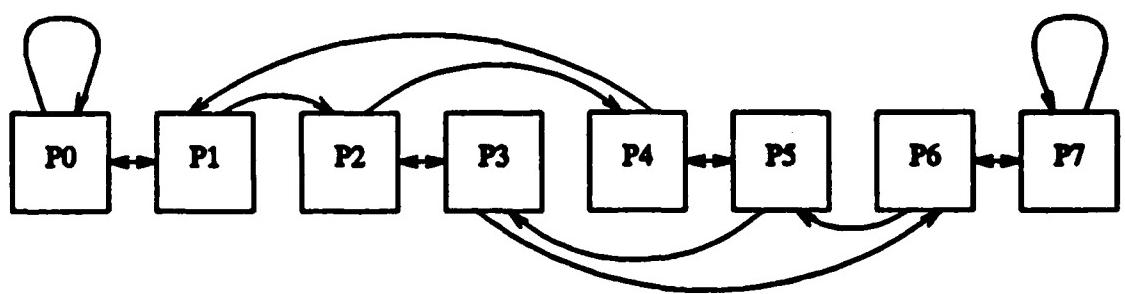


Figure 1.2.11. Perfect shuffle network for $N=8$.

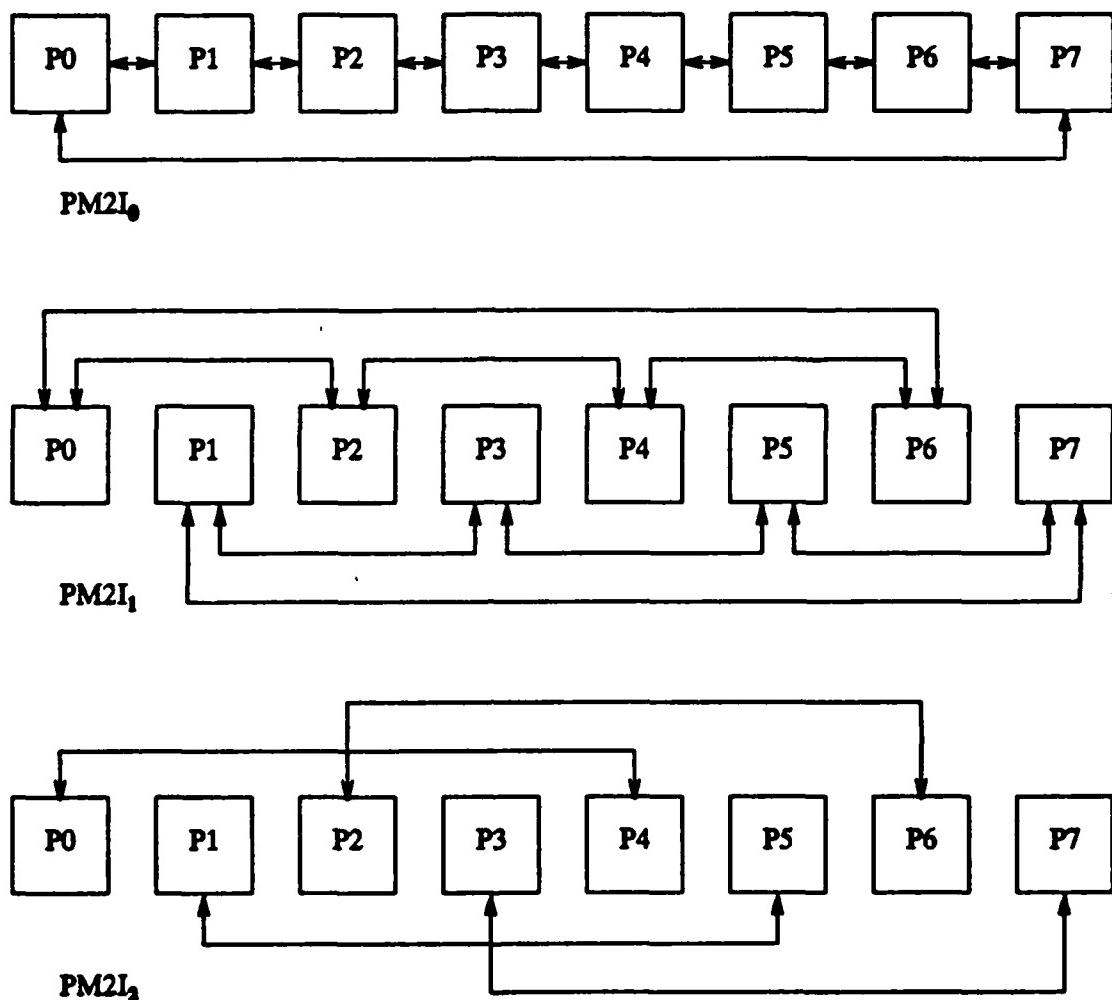
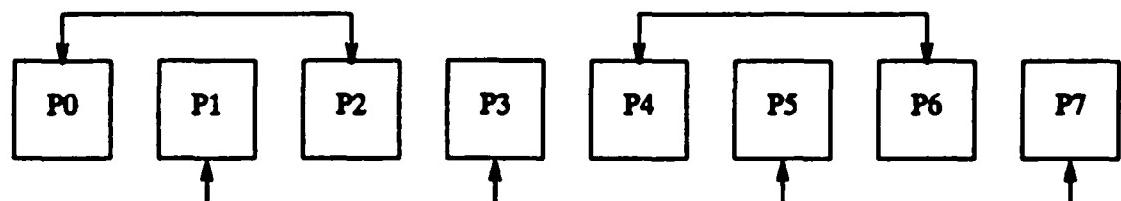
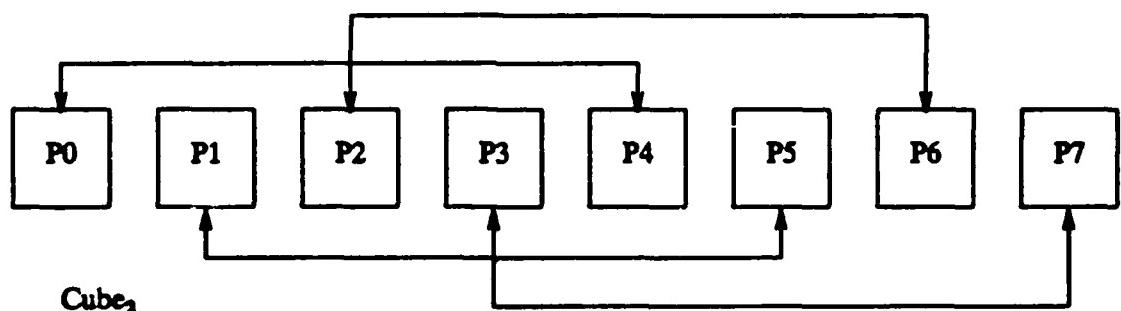


Figure 1.2.12. PM2I network for $N=8$.

The *Cube* network (Figure 1.2.13) has interconnection functions $\text{cube}_i(P)$ that map a processor address $P = p_{n-1} \dots p_{i+1}p_ip_{i-1} \dots p_0$ to the address $p_{n-1} \dots p_{i+1}\bar{p_i}p_{i-1} \dots p_0$, $0 \leq i < n$, resulting in a total of $Nn/2$ links. Conceptually, the processor addresses can be considered as the corners of an n -dimensional cube which connects each processor P to the n neighbors whose addresses each differ from P in one bit [Sie77a]. For this reason, this connection is often referred to as a *hypercube* network. Like the PM2I network, a maximum of n steps are required by any message. However, there is only $n-1$ -fault-tolerance. The Cube network is partitioned into two sub-networks by disallowing communication that uses any single cube function. The two partitions consist of those processors whose addresses match in the i 'th bit; processors whose addresses differ in the i 'th bit are prevented from communicating.

The preceding networks are all "single stage" or "recirculating" networks because processors are directly connected by a link and messages may have pass through the network (be recirculated) several times before they reach their final destination. To avoid the situation where messages must pass through intervening processors on their way to their final destination, multistage networks with specialized switching elements have been proposed and built. This frees the processors from the task of reading and forwarding messages. However, each message must traverse the multiple stages of the network completely; therefore, the length of time to send a message between any source-destination pair is fixed (typically at n steps).

A wired series of cube functions known as the Generalized Cube network is representative of the multistage Cube-type class of networks which include the Baseline [WuF80], Delta [Pat81], Extra Stage Cube [AdS82b], Indirect Binary n -Cube [Pea77], Omega [Law75], STARAN Flip [Bat76], and SW-Banyan ($S=F=2$) [GoL73]. The Cube has N inputs and N outputs. It is shown in Figure 1.2.14a for $N=8$. In a PE-to-PE machine configuration, processor P , $0 \leq P < N$, would be connected to input port P and output port P of the network. Data would flow unidirectionally from input to output. In a P-to-M machine configuration, processor P would be connected to input port P and memory P would be connected to output port P . Addresses would flow unidirectionally from input to output but data would flow bidirectionally: one way for reads, the other for writes. Because each processor has a single input

 Cube_0  Cube_1 Figure 1.2.13. Cube network for $N=8$.

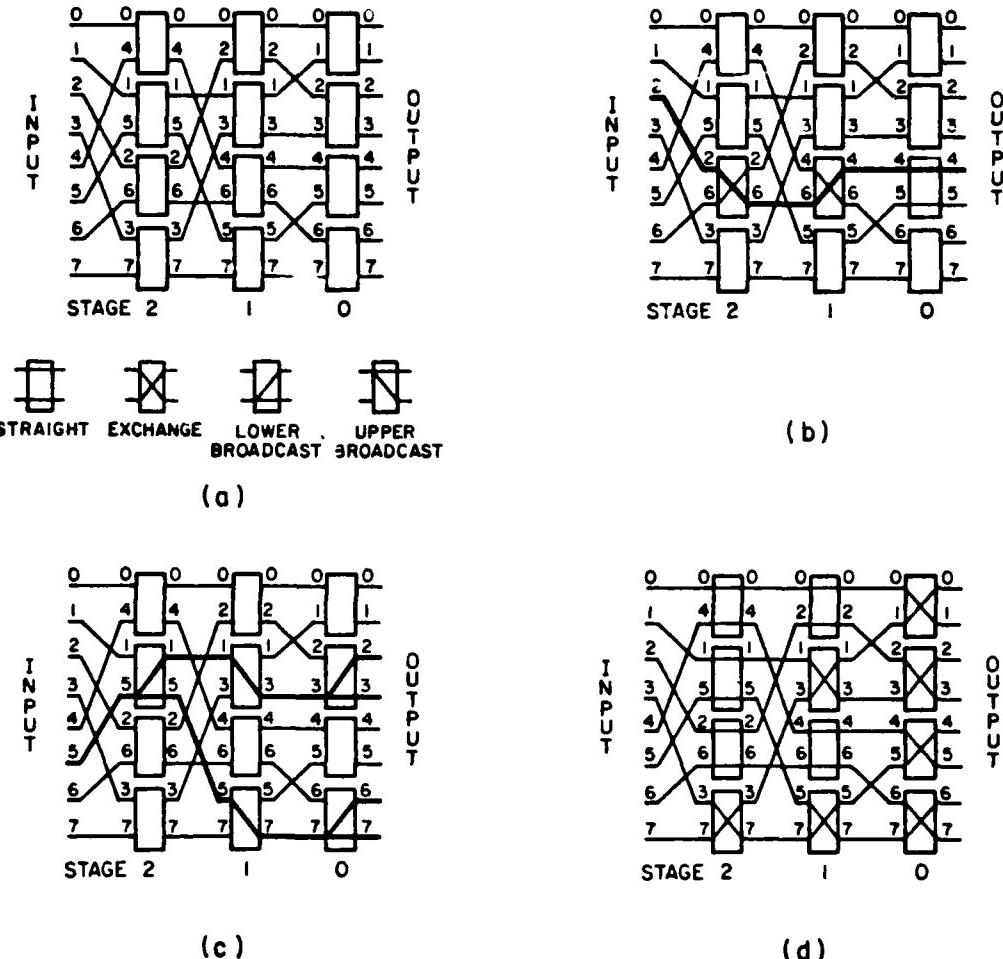


Figure 1.2.14. (a) Multistage Generalized Cube topology, shown for $N=8$.
 (b) Example one-to-one connection (input 2 to output 4). (c)
 Example broadcast connection (input 5 to outputs 2, 3, 6, and
 7). (d) Example permutation connection (input i to output
 $i+1$ modulo N).

connection to the network and only one available path exists between any source-destination pair (the destination being either another processor or a memory), the multistage Cube-type networks are not fault-tolerant.

The multistage Cube topology has n stages where each stage consists of a set of N lines connected to $N/2$ interchange boxes. Therefore, all messages require n steps to traverse the network regardless of the source-destination processor pair chosen. Each interchange box is a two-input, two output device. The labels of the input/output lines entering the upper and lower inputs of an interchange box are used as the labels for the upper and lower outputs, respectively. Stage i of the Cube topology contains the cube_i interconnection function, i.e., it pairs I/O lines that differ only in the i -th bit position. In the Generalized Cube network, each interchange box can be set individually to one of the four legitimate states shown in Figure 1.2.14a. Other networks in this class (e.g., n -Cube, Omega) are topologically equivalent, but may differ in their control schemes or may have different interchange box capabilities. A detailed discussion of the differences is given in [Sie79, Sie85]. Figures 1.2.14b, c, and d illustrate one-to-one, broadcast, and permutation connections, respectively. Note that many one-to-one and/or broadcast connections can occur simultaneously.

The network interchange boxes are controlled by routing or destination tags [Law75]. Both schemes allow network control to be distributed. In the routing tag scheme, the n -bit tag (T) for one-to-one connections is computed from the input port number (S) and desired output port number (D) using $T = S \text{ XOR } D$. Let $t_{n-1} \dots t_1 t_0$ be the binary representation of T . An interchange box at stage i need only examine t_i . If $t_i=1$, an exchange is performed; otherwise, the straight connection is used. In the destination tag scheme, $T=D$. If $t_i=0$, a connection is made from the interchange box input to the upper output link; otherwise, a connection is made to the lower output link. Thus, if the tag bits associated with a given interchange box are 0 on the upper input link and 1 on the lower input link, the box is set to the straight state. Similarly, if the tag bits are 1 on the upper input link and 0 on the lower input link, the box is set to the exchange state. Other combinations, e.g., 0 tag bit on both the upper and lower input, create "conflicts" in the network since no configuration of the box can make the desired connection [SiM80].

Tags that can be used for broadcasting data are an extension of this scheme [SiM81b]. An n -bit broadcast tag (B) indicates in what stages the boxes are to be placed in a broadcasting mode. For example, if bit B_i is a 1, a broadcast is performed; otherwise, the straight or exchange function specified by the normal tag T is used.

The multistage Cube network can be partitioned into independent subnetworks of various sizes such that all of the subnetworks have the same properties as the original. As discussed earlier, Cube networks can be partitioned based on any one of the n cube interconnection functions; the following example shows the partitioning based on the cube_0 function. A Cube network of size $N=8$ can be partitioned into two subnetworks of size four based on the low-order bit position such that one subnetwork connects only the even PEs and the other only the odd PEs. By setting all of the interchange boxes in stage 0 to straight, the two subnetworks are isolated. This is because stage 0 is the only stage which allows the even- and odd-numbered PEs to communicate. In general, a multistage Cube network can be partitioned such that all of the I/O ports in a partition of size 2^i agree in any $n-i$ bit positions.

One way to provide the multisage Cube-type networks with 1-fault-tolerance is to add an extra stage and to duplicate the input and output links. One example is the *Extra Stage Cube* network [AdS82b] (Figure 1.2.15) which is a fault-tolerant version of the multistage Cube network [DaS85a]. The network provides 1-fault tolerance because the extra stage at the input end of the network and bypass switches around stages n and 0 can be used to provide two disjoint paths through the network. Therefore, when a fault occurs, the alternate path may be used. This network has also been shown to be very robust under multiple faults [AdS84a].

There are two related multistage networks based on PM2I functions: the Data Manipulator (DM) [Fen74] and its enhancement, the Augmented Data Manipulator (ADM) [SiM81a]. As shown in Figure 1.2.16, the DM-type networks consist of $n+1$ stages of cells and n sets of links. In stage i , $0 \leq i < n$, PM2I_{+i} and PM2L_{-i} functions are implemented. There are also straight links between each stage of cells. Even though there are two distinct paths between each source-destination pair, there would not be any fault tolerance if processor P has a single output connection to cell P in stage $n-1$ or a single input

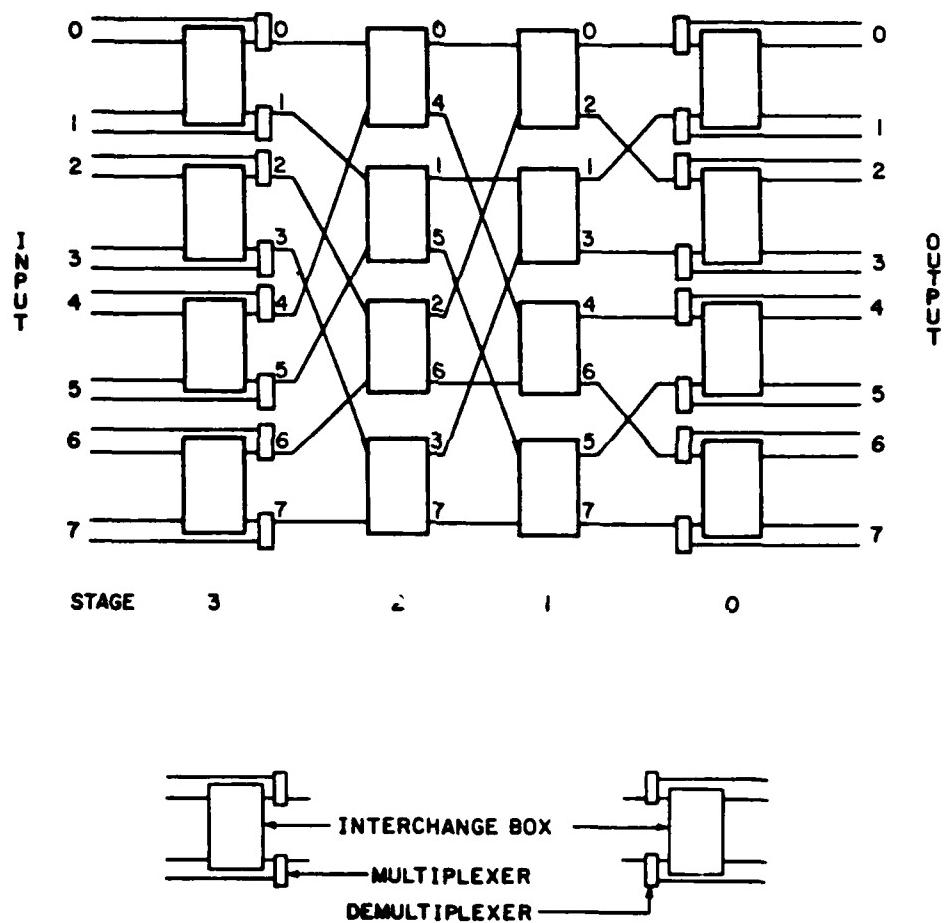


Figure 1.2.15. Extra Stage Cube network for $N=8$.

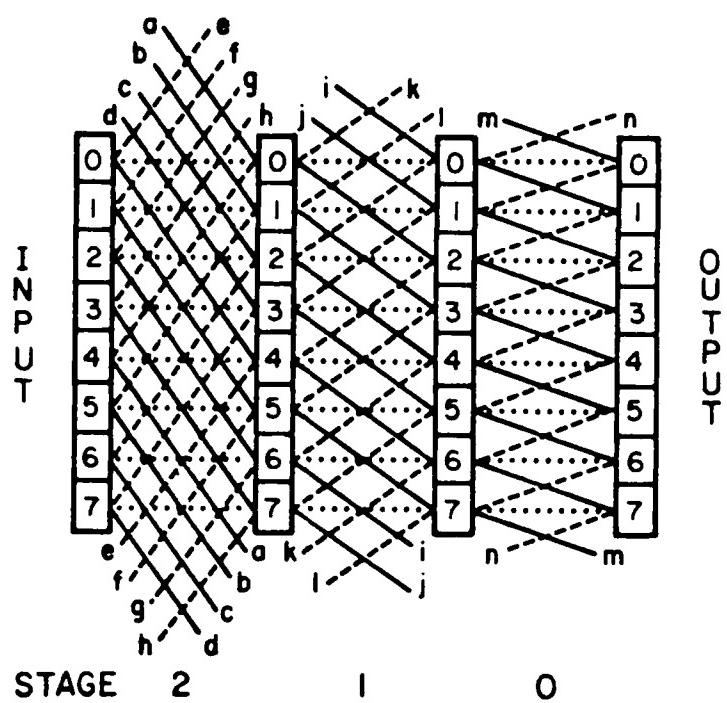


Figure 1.2.16. Data Manipulator network for $N=8$.

connection from cell P in stage 0. If bypass connections around stage $n-1$ and stage 0 were provided as in the Extra Stage Cube network, there would be 1-fault-tolerance.

The DM-type networks are more flexible than the multistage Cube-type networks in terms of the number of permutations that they can perform because of the presence of the straight links. However, this makes them more complex and hence, more costly. The ADM network is partitioned by requiring that the addresses of all of the I/O ports in a partition of size 2^i agree in their low-order $n-i$ bit positions. i.e., by forcing one or more of its low-order stages to "straight." In this respect, it is less flexible than the cube-type networks which could be partitioned on any set of $n-i$ bit positions. The control of DM-type networks is considered in [Sie85].

Circuit-Switched and Packet-Switched Networks

A *circuit-switched* network establishes a complete path (*virtual circuit*) between a source processor and the final destination processor. Such a path may span several links and pass through several switches. Once set, a switch is assumed to pass data from one of its inputs to one of its outputs and to induce a delay due only to the switching speed of the logic gates used to implement it. The switch may be set by tags (as in a multistage Cube-type network) or may be under software control (as in a single-stage recirculating network with intermediate processors).

In contrast, a *packet-switched* network establishes a path on a single link at a time. When a data *packet* (one or more words of data) completely arrives at an intermediate processor or interchange box, the incoming path is dropped and a new outgoing path is established to move the data one step closer to the final destination.

The choice of a circuit-switched or packet-switched network is dependent on many factors. The perceived advantages and disadvantages of each type are described below.

A circuit-switched network has the advantage that once the path through the network has been established and acknowledged, end-to-end data transmission is very fast. For an n -stage network implemented with two-input, two-

output switches, the round-trip delay is $2(n+1)T + 2nS_C$, where T is the link transmission time, and S_C is the delay time through a switch. A link is defined as a connection between the source PE and a switch in the first stage of a multistage network, between switches, or between a switch in the last stage of a multistage network and the destination PE. Since a switch establishes a "physical" connection between input and output, its delay is determined by the gate delays of the circuit elements within it.

Another advantage is that switching elements for circuit-switched networks are rather simple. They can be implemented with a two-by-two crossbar switch controlled by a finite state automaton having a small number of states. For this reason, circuit switching has the lower implementation cost.

By comparison, a two-input, two-output switch node in a packet-switched network requires two input and two output queues, each capable of holding one or more packets. The input queues contain packets that have arrived on the switch inputs that are waiting to be moved to one of the two output queues. The output queues contain packets waiting to be sent out of the switch to the next stage along an output link. These queues can consume a significant amount of chip area, especially if the packet size is large. Like the circuit-switching node, there is a two-by-two crossbar switch and a finite state automaton to control the arbitration of input and output links. However, the automaton is more complex because it is executed continuously and because it must deal with movement of packets between the input and output queues. Further, the movement of data from an input queue to an output queue is a serial process and cannot be done as fast as circuit-switched routing. The data movement requires a fetch from one of the two input queues, decoding of the datum's destination tag, and a store to one of the two output queues. The delay through the packet-switched network is given by $2(n+1)T + nS_p$, where S_p is the delay through one packet-switched stage. The $2(n+1)$ factor is due to the round-trip send-acknowledge cycle between network stages.

A disadvantage of circuit-switching is that it may take a long time to establish a path from a source to a destination PE since links along the entire path must be arbitrated. When links between switches are busy, other processors may be "locked out" of using certain interconnection paths. For the reason, response time may be poor.

In contrast, packet-switched response time is not greatly degraded for heavily-loaded systems because links between switches are individually secured and released. Thus there is a lower probability of a path being blocked by an existing path since packet switching uses only one link and two switches at a time instead of $n+1$ links and n boxes at a time for circuit-switching. This implies a shorter set-up time as seen by the processors.

Broadcast routings in circuit-switched networks automatically replicate the data as it passes through a switch. Unfortunately, the combination of the multiple destination acknowledgement signals into a single acknowledgement signal to be returned to the source is complex. The switches of packet-switched networks must explicitly replicate data for broadcast transfers by writing it to both output queues. Yet, broadcast communication is easy since each output link is arbitrated separately and the complex combination of acknowledgements is avoided.

For packet-switching, multiple packets all going to the same destination result in each link in the path having to be arbitrated for each packet. This arbitration is done only once per message in the circuit-switching case.

Since a data packet arriving at a destination processor can be from any source, recognition of the data requires that the packet's source tag be examined. Circuit-switching establishes the identity of the sender to the receiving processor at the time the link is established. This implies that the source tag needs to be examined only once per message for the circuit-switching case as opposed to once per packet for packet-switching.

Finally, arbitrarily large-sized blocks of data can be transferred between processors if circuit-switching is used. Only one network setting for the complete block needs to be performed. In contrast, packet-switching requires that the block be split into multiple packets. While the packets from a given source arrive at a destination in the order they were sent, the packets may be intermixed with those arriving from other sources. This necessitates the reassembly of packets into blocks on a source-by-source basis by the destination processor. It also defeats the possibility of an end-to-end block transfer being done with the aid of a Direct Memory Access controller or other block-oriented device.

1.2.8 Masking Schemes

In SIMD mode, all of the enabled processors execute instructions broadcast to them by their CU. A masking scheme is a method for determining which processors will be active at a given point in time. An SIMD machine may have several different masking schemes.

The *general masking scheme* uses an N-bit processor enable signal vector (a *general mask*) to determine which processors to activate. Processor i will be active if and only if the i -th bit of the processor enable signal vector is a 1, for $0 \leq i < N$. A mask instruction is executed by the CU whenever a change in the active status of the processors is required. Since the general mask is a part of the CU instruction stream, manipulation of such a mask is "reasonable" only for a limited N . For example, if the CU has 32-bit internal registers and a 32-bit data bus, only general masks for a machine with $N=32$ or smaller will be efficient to store and manipulate. When N is larger, say 1024, general masks become less appealing.

The *PE address masking scheme* [Sie77b] uses an n-position mask (a *PE address mask*) to specify which of the N processors are to be activated. Each position of the mask corresponds to a bit position of the processor addresses and consists of a 0, 1, or X (don't care). Processors whose addresses match the mask (0 matches 0, 1 matches 1, and 0 or 1 match X) are enabled. Square brackets denote a mask specification and superscripts are used as repetition factors. For example, $[X^{n-1}0]$ enables all even-numbered processors, while $[0^{n-i}X^i]$ enables PEs 0 to $2^i - 1$.

A PE address mask is fetched from the CU instruction stream and is decoded into a processor enable vector [SiS81b]. Also, several PE address masks can be decoded and their resulting processor enable vectors manipulated. For example, decoding two PE address masks, OR-ing them together, and using the result as the processor enable vector activates the union of the sets of processors activated by each individual mask [SiS81b].

Since PE address masks are a more compact notation than general masks, they are less costly to store and thus are more attractive for machines with a large number of processors. For example, a PE address mask for a 1024-processor SIMD machine consists of ten mask positions and can be encoded in 20 bits. (Two bits are needed per position to encode 0, 1, or X). The decoding

of a PE address mask into a general mask and the manipulation of general masks is typically done by special-purpose hardware external to the CU CPU. This reduces the need for extremely large registers and data paths within the CU CPU itself.

Data conditional masks are the result of performing a test on local processor data, where the results of different processors' evaluations may differ. They are used when the decision to enable and disable processors is made at execution time. As a result of the parallel equivalent of the if-then-else conditional ("where statement" [SiS81b]):

if (<parallel-expression>) then <statement> else <statement>

each processor will evaluate the expression to determine if it should be enabled for the "then" part or the "else" part. The result is an N-bit data conditional mask comprised of N one-bit "true/false" data conditional results, one result from each processor. The "true/false" data conditional results are stored for use in activating or deactivating the processors. A stack can be used for storing masks associated with nested conditionals.

Certain CU CPU instructions cause a branch based on the results of data conditional masks. For example, "if any" processor meets some criteria (a bit in the data conditional mask is "true"), the CU would execute a branch to a different part of the SIMD program.

Data conditional masks can be used to simulate the application of a general mask. Suppose the CU broadcasts instructions to the processors to evaluate the expression " $P < N/2$ " (where P is a processor's number). Processors numbered 0 through $(N/2)-1$ would find the expression to be true while the rest would find it to be false. The resulting data conditional mask is identical to a general mask where the first $N/2$ bits were set.

Since the application of masks controls the state of the processors and may affect the program flow, all types of masks are manipulated by the CU, either directly or indirectly. One difficulty with allowing the processors to manipulate masks is that once a processor disables itself, it cannot perform instructions to re-enable itself. Some SIMD machines overcome this by allowing the CU to temporarily force all of the processors to an enabled state. Others avoid the difficulty by using the CU to store and manipulate all types of masks itself, using the processors only to calculate data conditional masks.

CHAPTER 3

SURVEY OF RELATED LITERATURE

Among the variety of high-performance architectures that have been proposed or constructed, there is one common characteristic: the use of parallelism to increase performance. In this chapter, some examples of each of the models of parallelism given in the previous chapter are presented.

The brief synopses of the small number of parallel computers summarized in this chapter cannot possibly do justice to those computers' sophisticated designs or to the variety and diversity of parallel computer architectures and applications. Therefore, the author has chosen systems for this review that are particularly well-known, successful, novel, or of historical importance. This review serves only to introduce the systems that are referred to later in this thesis and to provide references where additional information on the systems can be obtained. Specific design aspects of these machines are considered in detail elsewhere in the thesis when related design decisions for PASM are being discussed.

1.3.1 SIMD Machines

Illiac IV

The first major SIMD machine to be constructed was the Illiac IV [BaB68, BoD72]. It was based on earlier proposed SIMD machines, notably the Unger and Solomon [SLB62] machines. The Illiac IV had 64 PEs, each capable of performing several types of integer and floating point operations on its internal 64-bit words. The PEs were interconnected in a (PE-to-PE) four-nearest-neighbor pattern: PE P had a direct connection to PEs numbered $P \pm 1$ (modulo 64) and $P \pm 8$ (modulo 64). Each PE had a 2K 64-bit word semiconductor RAM memory which was also accessible to the CU.

An extremely sophisticated CU fetched SIMD instructions from the PE memories, executed the control flow operations (loop counting, branching, etc.), and placed the arithmetic (PE) operations to be executed into a FIFO queue. The CU employed an instruction cache to increase performance by reducing the accesses it made to the PE memories for instructions.

A special CU subunit decoded the PE instructions into control signals and placed these on an instruction bus where they were broadcast to the PEs. A 64-bit general mask vector maintained in a special CU register indicated which PEs were enabled and disabled. Resetting this register changed the enabled status of the PEs [Ste75]. A copy of this register was stored in the PEs, one bit per PE. Most instructions broadcast by the CU would not be executed if this bit was not set. However, certain PE instructions forced all of the PEs to execute, regardless of their enabled/disabled state. In addition, a 64-bit data conditional mask formed from the "mode bit" line from each PE could be read by the CU into any one of its internal registers.

Besides its historical importance as the first operational SIMD machine, the Illiac IV also was the first to make exclusive use of semiconductor RAM for its primary memory [Fal76] and the first to use a queue to overlap the operations of the CU and PEs. Its other major contribution is its use of multiported PE memories that allowed concurrent access by the associated processor, the CU, and the I/O subsystem. Illiac IV was dismantled in 1981.

BSP

The Burroughs Scientific Processor (BSP) [Bur77, KuS82] was developed by the Burroughs Corporation during the late 1970s. It had 16 48-bit processors and 17 memory modules arranged in a P-to-M configuration. This organization provided conflict-free memory access to vectors of arbitrary length with a stride (logical distance between selected vector elements) not equal to 17. It is this memory organization and the associated study of skewed array storage techniques that is the BSP's principal innovation. Burroughs discontinued development of the BSP in 1979 without ever bringing it to market.

PEPE

PEPE [CrG72, Ens74, ViC78] has 288 PEs, each consisting of an arithmetic unit, correlation unit, output unit, and a 1K by 32-bit memory. It is an example of a word-associative processor without an interconnection network. The CU is a full-fledged word-associative processor itself, capable of simultaneous associative, arithmetic/logic, I/O, and control functions. PEPE was used for real-time radar tracking applications [Wil72, Cor72].

Each PEPE PE has a three mode bits that independently control the enabled/disabled status of its arithmetic, correlation, and output units. Of course, some instructions are performed by the PEPE PEs regardless of the setting of the mode bits (such as resetting the mode bits themselves). Data conditional masks can be formed, but they are not communicated to the CU directly. Instead, the CU is interrupted "if any" PE has found a condition to be true and is given the address of the lowest-numbered processor that has found the condition to be true. Repetitive requests for this address can be used by the CU to determine all of the processors for which a data condition was true. This "first responder" scheme is typical of associative processors such as PEPE.

STARAN

STARAN [Bat74, Bat77b] is a bit-associative processor having up to 32 associative array modules, each of which contains a 256-word by 256-bit multidimensional access memory [Bat77a], 256 bit-serial processors, and a STARAN Flip network [Bat76]. Within an array module, the unidirectional network is connected to the processors and memories such that either can act as a source or destination for data; therefore, STARAN implements both a P-to-M and a PE-to-PE computation model. (The STARAN PEs have a local "memory" of a few fast-access registers). The Flip network allows the access of memory in bit or word slices or a combination of the two modes [Bau74].

Like the Illiac IV PEs, each STARAN processor has a single mode bit that controls its enabled/disabled status. Of course, some instructions are performed by the STARAN processor regardless of the setting of the mode bit. Like PEPE, data conditional masks can be formed, but the 256 result bits are not available to the CU directly. Instead, the CU receives the OR of the bits which can be used to determine the "if any"-type conditions. Also, the CU can receive the address of the lowest-numbered processor that has contributed a "1" to the data conditional mask. Repetitive requests for this address can be used by the CU to determine all of the processors for which a data condition was true.

A STARAN computer also has a Parallel Input/Output (PIO) module which permutes data between eight 256-bit ports and which itself contains a flip network. The PIO module ports can be connected to associative array modules, to external I/O devices, or to other PIO modules. Multiple PIO modules are used to allow communication among all (as many as 32) associative array modules. Principally used for image processing [RoP77, Kry76, Dav74, VoF77], its major contribution is the use of multistage interconnection networks that allow any processor to fetch any data item from the memory using a single instruction. The PIO modules also foreshadowed the use of multistage permutation networks for I/O in later machines.

CLIP-4

CLIP-4 [Duf76, Fou81] is a bit-slice array processor of 96-by-96 PEs with an 8-nearest-neighbor interconnection pattern. Its principal importance is its demonstration of the use of truly large numbers of processors for image processing applications [Duf85]. While a 32-associative-array-module STARAN system has nearly as many processors as does CLIP-4, communication among STARAN associative array modules is indirect through the PIO modules. This made STARAN somewhat less practical than CLIP-4 for algorithms and images in which communication traffic spanned the boundaries of array modules.

MPP

The Massively Parallel Processor (MPP) [Bat82] is a 128-by-128 bit-slice array processor with a 4-nearest neighbor interconnection pattern. Its control and masking schemes are quite similar to those of STARAN. MPP's most distinguishing feature is a staging memory interfacing a high-speed I/O bus with the PE array [Bat84]. The staging memory is used to reformat and reshuffle data among the various bit planes in the MPP memory system, thus attacking two of the principal limitations of the earlier SIMD architectures: providing adequate I/O bandwidth and providing a flexible external (not involving the processor array) data formatting and preparation facility. The staging memory is an important improvement on the STARAN PIO modules. In STARAN, the PIO modules had no memory; therefore, the associative array memory had to be used to store temporary data during reformatting and reshuffling operations.

1.3.2 MIMD Machines

Many examples of machines capable of operating in MIMD mode are currently available since even a single-bus shared-memory system with two CPUs qualifies as a bona-fide MIMD machine. The systems that will be presented here, however, are those in which multiple processors commonly cooperate on a single user-level application task. This differs from the more common case of a set of processors cooperating at the operating-systems level, passing messages to each other (in mail handling, for example) or in

distributing complete user tasks among the processors for load balancing purposes.

C.mmp

An example of a tightly-coupled system MIMD system is C.mmp [WuB72], now dismantled, which was composed of 16 PDP-11/40 processors connected to 16 shared memory modules via a 16-by-16 crossbar switch. Its importance is partly historical due to the large number of processors employed (for the early 1970s), but mostly it is recognized for the contributions of its operating system, Hydra [WuC74], to the area of distributed operating systems research.

Cm*

Another research multiprocessor system, Cm* [SwF77, SwB77], is a loosely-coupled hierarchical system containing 50 LSI-11 processors. A Cm* "computer module" consists of a CPU, memory, and local I/O. Several computer modules are arranged on a "Map bus" to form a cluster. Data transfers between computer modules in the same cluster are handled on the Map bus. Processors in different clusters communicate via "intercluster buses" which are connected to the Map buses through a special mapping device known as a "Kmap." Like C.mmp, its principal notoriety is due to its use of a significant number of processors and the innovativeness of its Medusa operating system. It is also important because it explored the effectiveness of interconnecting the processors in a hierarchical structure.

ZMOB

ZMOB [KuW81, KuW82] is a collection of 256 PEs arranged on a ring network developed at the University of Maryland. Each PE consists of a Zilog Z80A 8-bit microprocessor, a 64K-byte local memory, floating point unit and hardware multiplier, serial and parallel I/O ports, and an interface to the ring network. The ring network "conveyor belt" is a 48-bit-wide shift register with 257 stages or "mail stops." PEs occupy 256 of the mail stops; the remaining

stop is used by the host computer. ZMOB is intended mainly for image processing applications.

HEP

The Heterogeneous Element Processor (HEP) [Jor83] is a commercially-available multiprocessor system developed by Denelcor, Incorporated. It can have as many as 16 processors and 128 data memory modules connected in a P-to-M configuration by a synchronous packet-switched network. Each processor is a sophisticated, highly pipelined unit with a local program memory. The network consists of an arbitrary number of three-ported nodes that are used to interconnect processors, memories, I/O devices, and other nodes in whatever pattern is desired. A typical arrangement clusters a processor with one or more memory modules, and connects these clusters together using a hypercube topology. Clusters are often connected with the I/O subsystem in a tree topology. HEP's major contribution was the demonstration of the use of a packet-switched interconnection network with distributed routing tables in a large system. It is also important historically as the first commercial large-scale multiprocessor.

Intel iPSC Hypercube

The Intel iPSC consists of up to 128 nodes, each consisting of an Intel 80286 CPU, 80287 floating point co-processor, local memory, and interconnection network ports. It is a message-passing, non-shared-memory system in which the nodes are interconnected in a hypercube topology. A drawback to its design is that there is a single channel between the set of nodes and the secondary storage devices; therefore, loading and unloading the data is time-consuming. Because it uses no custom-designed components, it is a relatively low-cost machine that has enjoyed considerable commercial success.

BBN Butterfly

The BBN Butterfly machine [CrG85, Ret83] consists of up to 256 "Processor Nodes," each consisting of a Motorola MC68000 CPU, a co-processor for memory management, memory, and a connection to a multistage Cube-type packet-switching network. The machine is being developed by Bolt, Beranek, and Newman, Incorporated (BBN). It is a tightly coupled multiprocessor in which part of each nodes' physical memory appears in a single shared address space. CPU memory references are constantly monitored by the co-processor and references to memory addresses local to the node are satisfied immediately. Non-local references are trapped by the co-processor, a "data request message" is dispatched to the remote node through the network, the co-processor of the remote node retrieves the data, and the data is returned via the network to satisfy the reference.

The Butterfly machine design enjoys a speed advantage over a strict shared-memory system since local references need not go through the interconnection network and global (shared) references need not be encapsulated in messages. Therefore, it can take advantage of program and data reference locality [Den70] so long as the programmer or compiler locates the code and the data it operates upon in the same processor. The Butterfly machine demonstrated the first use of a multistage interconnection network in a large-scale MIMD system.

Ultracomputer

The NYU Ultracomputer [GoG83] is a shared-memory P-to-M machine configuration with a multistage message switching network. A system of up to 4096 PEs has been proposed by researchers at New York University (NYU) and a 512-PE prototype is being constructed as a joint venture between NYU and International Business Machines, Incorporated (IBM). The IBM-constructed version is known as the "Research Parallel Processor Prototype" (RP3). Like the BBN Butterfly machine, each RP3 PE consists of a CPU, a co-processor for memory management, memory, and an intelligent network connection to a multistage Omega-topology packet-switching network. To increase performance, each PE also has a local cache memory. RP3 offers additional

flexibility over the Butterfly in that the memory associated with each PE can be partitioned under software control. The memory can be made to appear to be local to the PE, shared among all PEs, or a combination of these two. Therefore, RP3 can act as a strict local-memory system with message-passing, a strict shared-memory system, or a hybrid.

Another novel feature of the Ultracomputer/RP3 design is the implementation of the "Fetch-and-Add" synchronization primitive in the interconnection network hardware [GoG83]. This synchronization mechanism allows simultaneous read-modify-write access to a single shared variable by multiple processors without serialization. Its use and importance will be discussed in the later chapters dealing with operating systems and synchronization.

Trends in MIMD Design

While there are other high-performance commercial multiprocessors capable of MIMD operation, the processors of such systems are not typically used to jointly work on a single task. Such systems either have a very small number (two to four) highly pipelined CPUs and shared memories (e.g., Cray X-MP, Cyber 170) or a more moderate number (up to sixteen) number of more conventional CPUs which communicate over a high-speed bus (e.g., Sequent Balance). With suitable operating system enhancements and parallel application algorithms, the processors of these machines could cooperate on a single task.

The recent flurry of activity in the design and construction of multiprocessors is due in part to the difficulty in using SIMD architectures. Efficient SIMD programs are, in general, more difficult to construct and require specialized languages. On the other hand, MIMD operation can be specified with conventional computer languages with the addition of operating system calls for forking and joining processes [Dij68], using semaphores [Dij68], and accessing shared data structures. The recent trend seems to be toward the use of very large numbers of simple microprocessors interconnected by multistage networks rather than the use of smaller numbers of cooperating supercomputers.

1.3.3 MSIMD and MMIMD Machines

MSIMD machines are a special class of SIMD machines in that they support several instruction streams; however, each instruction stream is associated with an independent SIMD process. The original Illiac design was for a 4-CU, 256-PE system capable of operating in MSIMD mode. Later, MAP [Nut77a, Nut77b] was proposed. It was to be an 8-CU, 1024-PE system. Neither system was ever constructed: the remaining three 64-PE "quadrants" of the Illiac configuration were never developed due to lack of funds and the difficulty of implementing an appropriate CU-PE switch in MAP was its primary drawback. In addition, MSIMD machines require a partitionable interconnection network to prevent interference from competing SIMD processes.

Caltech Cosmic Cube

The 64-processor Cosmic Cube (Mark I) [Sei85] and its enhancement, the 128-processor Caltech/JPL Mark II Hypercube [TuP85], are quite similar to the Intel iPSC design. One difference is that the less-sophisticated Intel 8086/8087 pair is used in the Cosmic Cube nodes. In another way however, the Cosmic Cube is an improvement over the Intel iPSC since groups of 32 nodes can be independently controlled, allowing MMIMD operation. Also, multiple secondary storage systems are used to improve I/O bandwidth: one disk is associated with each set of 32 nodes.

1.3.4 SIMD/MIMD Machines

The idea of combining the SIMD and MIMD modes of operation in one system was originally considered by Lipovski and Tripathi [LiT77]. Such a machine could use the same processors to perform the computations in both modes of parallelism so that data need not be moved from computer system to computer system. Also, the algorithms making up a complete task may be coded to utilize the most efficient mode of parallelism.

TRAC

[SeU80] is an experimental SIMD/MIMD computer system in a P-to-M configuration where the processors and memories (or I/O devices) are interconnected with a SW-Banyan-type multistage network [GoL73]. The SW-Banyan network allows the number of processors to be different (typically smaller) than the number of memories and I/O devices. A 4-processor, 9-memory (or I/O) prototype is under construction at the University of Texas at Austin.

TRAC operates in MIMD mode in much the same way as the NYU Ultra-computer: each processor uses the network to access data memories or I/O devices. Processors communicate by placing messages in shared memory areas. In SIMD mode, a processor is chosen as the control unit since there are no dedicated control units in TRAC. The control unit sets the network in a "data tree" configuration by connecting itself to one or more memories where the SIMD instructions are stored. The TRAC "tree" is so named because the set of network paths between the control unit (at the root of the tree) and the memories is conceptually a tree structure. The CU processor then sets up an "instruction tree" on which it broadcasts the instructions to the set of processors performing the computations. Finally, these processors set up "data trees" or "shared memory trees" by setting network switches to connect themselves to the data memories.

PASM

The main focus of the work described in this thesis is PASM, a partitionable SIMD/MIMD processing system [SiS81b]. It uses the PE-to-PE computation model, employs microprocessor-based PEs interconnected by a multistage network, and has dedicated SIMD control units and other system control and memory management processors. It will be discussed in detail in the next section.

CHAPTER 4

PASM OVERVIEW

1.4.1 Basic Components

A block diagram showing the basic components of PASM [SiS81b, SiS84] is given in Figure 1.4.1. The *System Control Unit* is a conventional computer and is responsible for the overall coordination of the activities of the other components of PASM. The *Parallel Computation Unit* contains $N=2^n$ processors, N memory modules, and a multistage interconnection network. The *Memory Management System (MMS)* controls the loading and unloading of the Parallel Computation Unit memory modules from the multiple secondary storage devices of the *Memory Storage System*. The *Micro Controllers (MCs)* are a set of Q microprocessors which act as the control units for the PEs in SIMD mode and orchestrate the activities of the PEs in MIMD mode. Each MC controls N/Q PEs. An MC together with its N/Q PEs is called an *MC-group*. *Control Storage* contains the programs for the MCs.

PASM is being designed to have $N=1024$ PEs and $Q=32$ MCs. The PASM prototype, illustrated in Figure 1.4.2, is being constructed with $N=16$ PEs and $Q=4$ MCs.

1.4.2 Parallel Computation Unit

The PE processors are microprocessors that perform the actual SIMD and MIMD computations. The N PEs are numbered from 0 to $N-1$ and each PE knows its number. PE memory modules are used by the processors for data storage in SIMD mode and for both data and instruction storage in MIMD mode. Each PE can operate in both the SIMD and MIMD modes of parallelism. A pair of memory units is used for each memory module to allow data to

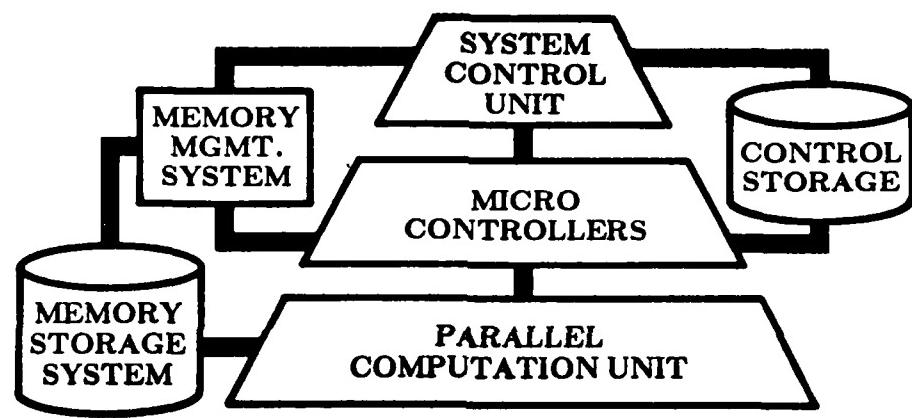


Figure 1.4.1. PASM block diagram.

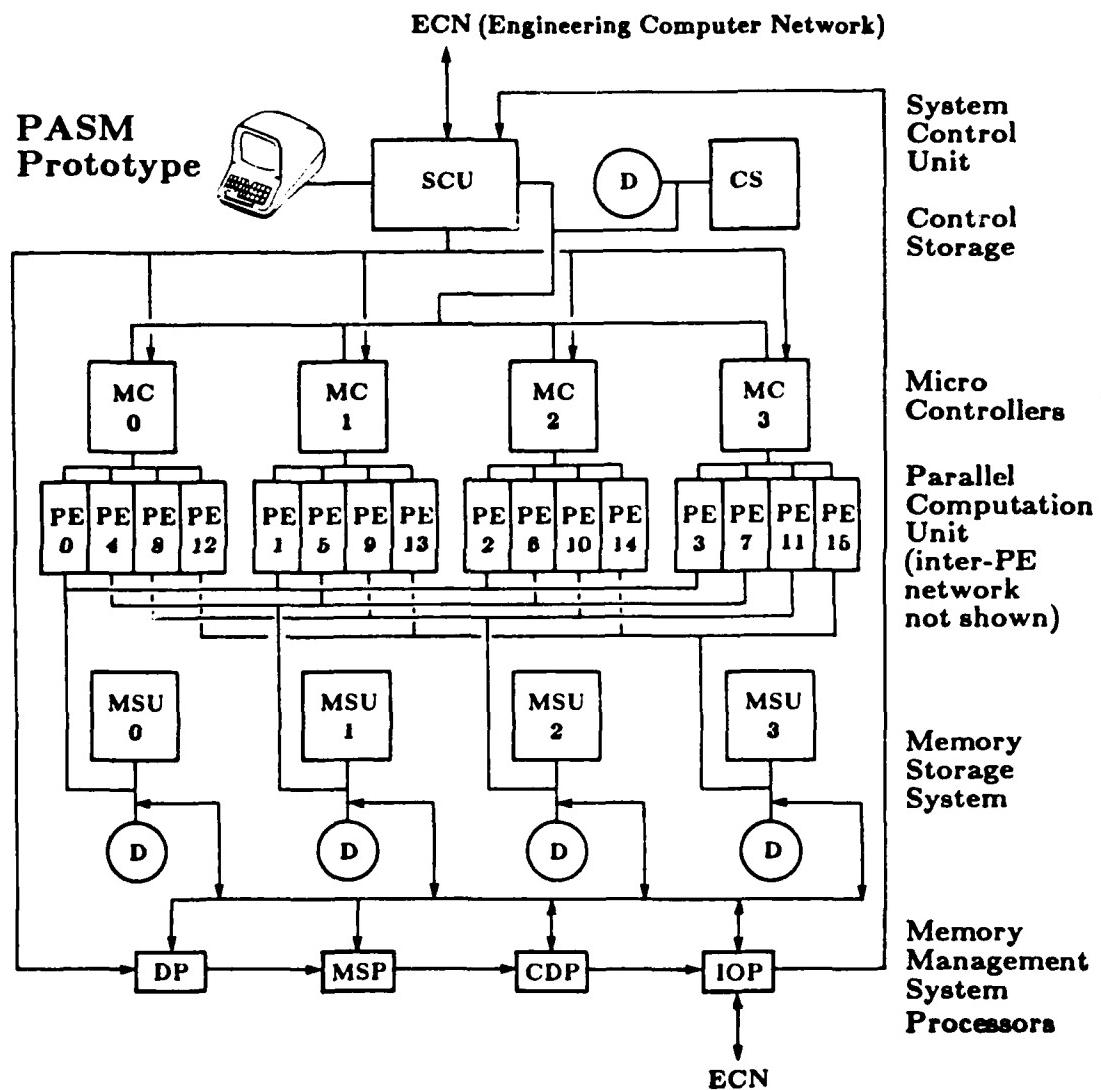


Figure 1.4.2. PASM prototype overview.

be moved between one memory unit and secondary storage while the processor operates on data in the other memory unit.

The interconnection network provides a means of communication among the PEs. Two types of multistage interconnection networks have been considered for PASM: the Generalized Cube [SiM81b] and the Augmented Data Manipulator (ADM) [SiM81a]. The characteristics and operation of these networks were described in an earlier section. The partitionability of these networks is an important property because it allows the set of PASM PEs to form partitions of various sizes. Since the multistage Cube can be partitioned such that all of the I/O ports in a partition of size 2^i agree in any $n-i$ bit positions, while the ADM requires that the I/O ports agree in the low-order $n-i$ positions, the ADM is the more restrictive of the two networks in this respect. Therefore, to allow either network to be used, PASM's partitionability will be limited to that which the ADM can perform. The ramifications of this decision and additional details relating to the network's operation in PASM are given below.

1.4.3 MCs and Partitions

The MCs are the multiple control units needed in order to have a partitionable SIMD/MIMD system. There are $Q=2^q$ MCs, physically addressed (numbered) from 0 to $Q-1$. As discussed earlier, the physical addresses of the N/Q processors which are connected to an MC must all have the same low-order q bits so that the network can be partitioned. The value of these low-order q bits is the physical address of the MC. An SIMD machine partition of size MN/Q , where $M=2^m$ and $0 \leq m \leq q$, is obtained by having M MCs use the same instructions and synchronizing the MCs. The physical addresses of these MCs must have the same low-order $q-m$ bits so that all of the PEs in the partition have the same low-order $q-m$ physical address bits. Similarly, an MIMD machine partition of size MN/Q is obtained by combining the efforts of the PEs associated with M MCs which have the same low-order $q-m$ physical address bits. In MIMD mode, the MCs are used to help coordinate the activities of their PEs. Q is the maximum number of partitions allowable, and N/Q is the size of the smallest partition. Each MC has two memory units so that memory loading and computations can be overlapped.

In SIMD mode, each MC fetches instructions from its memory module, executes the control flow instructions (e.g., branches) and broadcasts the data processing instructions to the PEs in its MC-group. An *Instruction Broadcast Bus* is shared by the MC and the set of PEs so that all of the PEs receive the same instruction at the same time. In MIMD mode, the MCs may use the bus to communicate synchronizing instructions to the PEs. An additional *MC-PE Communication Bus* is used in both SIMD and MIMD modes. Individual PEs use the bus to alert their MC to any internal PE fault or error (e.g., memory fault, arithmetic error).

The PEs of a partition of size MN/Q are logically addressed from 0 to $(MN/Q)-1$. Specifically, the logical address of a PE is given by the $m+n-q$ high-order bits of its physical address. While the physical address of a PE is fixed, its logical address varies with the size of the partition in which it is participating. The PASM operating system converts logical PE addresses to physical ones based on the partition chosen to perform a program. Therefore, a system user will deal only with logical addresses in a program.

The approach of permanently assigning a fixed number of PEs to each MC has the advantages that the operating system need only schedule Q MCs (rather than N PEs) and that the MC/PE interaction is simplified when a partition is formed. Also, this fixed connection scheme allows the efficient use of multiple secondary storage devices as discussed later.

The *designator* of a partition is the smallest physical address of an MC involved in the partition. This designator corresponds to the low-order $q-m$ bits of the physical address of any MC participating in the partition [TuS82b]. The *designated MC* is the MC with the designator.

In SIMD mode, all of the enabled PEs within a partition execute instructions broadcast to them by their MC(s). Because an MC controls only N/Q PEs, a single MC need only fetch and manipulate masks of N/Q bits. If general masks are stored in the SIMD program, they will consist of a total of MN/Q bits; however, any single MC will need to fetch and manipulate $1/M$ th of the bits. Because N/Q is generally of the same magnitude as the machine register and bus sizes, PE address masks can be decoded in software, if necessary, and their resulting PE enable vectors manipulated. In addition, each MC receives only N/Q data conditional mask bits. Therefore, these can be

examined and manipulated by the CU CPU as well. One drawback to having each MC receive only N/Q data conditional mask bits is that for partitions involving more than one MC, MCs must communicate to determine the global status of the PEs. The global status is necessary to evaluate the "if-any"-type conditions. The mechanisms for this communication will be discussed later.

1.4.4 Secondary Storage

Control Storage (CS) contains the programs for the MCs. The loading of programs from Control Storage into the MC memory units is controlled by the System Control Unit.

The *Memory Storage System* provides secondary storage space for the PE data files in SIMD mode and for the PE data and program files in MIMD mode. It consists of N/Q independent *Memory Storage Units (MSUs)*, numbered from 0 to $(N/Q)-1$. Each MSU consists of a high capacity disk drive, disk controller, and a microprocessor to manage the file directory system on the disk. Each MSU is connected to Q PE memory modules. For $0 \leq i < N/Q$, MSU i is connected to those PE memory modules whose physical addresses have the value i in their $n-q$ high-order bits. Recall that, for $0 \leq k < Q$, MC k is connected to those PEs whose physical addresses have the value k in their q low-order bits. This is shown for $N=16$ and $Q=4$ in Figure 1.4.2.

A partition of MN/Q PEs requires only M parallel block loads if the data for the PE memory module whose high-order $n-q$ logical address bits equal i is loaded into MSU i [SiS81b]. This is true no matter which group of M MCs (which agree in their low-order $q-m$ physical address bits) is chosen. Each PE has two memory units to allow data to be moved between one memory unit and secondary storage while the processor operates on data in the other memory unit.

1.4.5 Memory Management System

The *Memory Management System* controls the transferring of files between the Memory Storage System and the PEs. It is composed of a separate set of microprocessors dedicated to performing tasks in a distributed fashion. This

distributed processing approach is chosen in order to provide the Memory Management System with a large amount of processing power at low cost. The Memory Management System (Figure 1.4.2) consists of a Directory Lookup Processor (DP), a Memory Scheduling Processor (MSP), a Command Distribution Processor (CDP), and an Input/Output Processor (IOP). The division of tasks chosen is based on the main functions which the Memory Management System must perform, including: (1) generating tasks based on PE load/unload requests from the System Control Unit; (2) scheduling Memory Storage System data transfers; (3) controlling input/output operations involving peripheral devices and the Memory Storage System; (4) maintaining the Memory Management System file directory information; and (5) controlling the Memory Storage System bus. User programs and data can be received from or sent to peripherals such as additional mass storage or image input/output devices.

1.4.6 System Control Unit

The *System Control Unit (SCU)* is responsible for the overall coordination of the activities of the other components of PASM. The types of tasks the SCU will perform include program development, job scheduling, and coordination of the loading of the PE memory modules from the Memory Storage System with the loading of the MC memory modules from Control Storage.

CHAPTER 5

PASM HISTORY

In this chapter, a chronological history of the development of the PASM architecture and its prototype implementation is given. This information is included to allow the interested reader to explore early PASM design and implementation ideas and to find details and analyses not central to the current PASM design effort.

In August 1978, the first paper describing the PASM architecture was presented at the International Conference on Parallel Processing [SiM78b]. It contained most of the major elements of the current PASM design: Q MCs, N PEs arranged in a fixed pattern of N/Q PEs per MC, a multistage network, a distributed Memory Management System, and a System Control Unit. The only notable omission was the scheme for connecting the Memory Storage Units to the PEs allowing the block loading of the PEs within a partition. Inter-MC communication, enabling/disabling of the PEs, and other control schemes are all described, although the implementations proposed are not necessarily those that would be proposed today due to technological progress. Another difference was that the paper proposed that only a subset of the PEs were to be able to operate in MIMD mode.

One year later, at the 1979 International Conference on Parallel Processing, the parallel primary/secondary memory system for PASM was described [SiK79]. This essentially completed the study of the PASM "topology" which has remained unchanged since then. However, several implementation ideas proposed in these early papers are now obsoleted by changes in processor, interconnection, and memory technology.

In December 1981, the PASM architecture ideas and some examples of parallel image processing algorithms were published in the IEEE Transactions

on Computers [SiS81b]. This article included one new architecture concept, a "reconfigurable shared bus" for connecting MC CPUs to MC memory modules allowing the sharing of SIMD program code among MCs and giving a degree of fault tolerance. This shared bus concept will be described in the next section. Also at that time, the choice of the PASM interconnection network had been narrowed to two types: the multistage Cube [SiM81c] and the ADM [SiM81a].

Steps toward a prototype of PASM were taken as early as 1979. From unpublished notes sketched by Robert McMillen, a 16-PE, 4-MC, 4-MSU system was proposed and some implementation details were specified. The proposal used microprogrammable bit-slice CPUs as MCs and PEs and employed commercial disks with modified disk controllers as MSUs. A commercial 8-bit microprocessor board was proposed for the SCU. Memory management system processors were also 8-bit CPUs, but the proposal did not include information about their roles or the specific interconnections between them. Because no funds for construction were available at the time, this proposal was never implemented.

It is not surprising that the first prototype proposal called for customized PEs and MCs since all previously-constructed SIMD machines had specialized PE and CU designs. [A "customized" processor is one whose instruction set and set of input/output signals is specified by the customer for a specific application. It may be a specialized LSI design or may be implemented with commercially-available components.] In [SiM78b], a design proposing a PE capable of decoding its own instructions in MIMD mode and accepting "micro-code" control signals produced by the MC in SIMD mode was put forward. Such a design would have required custom MC and PE processors. The specialized PE instruction set proposed in [SiM78a] helped to further the conception that a custom processor had to be constructed. In the context of the time period, the custom approach seems suitable: technology at the time had bit-slice architectures operating at several times the clock speeds of commercial 8-bit microprocessors.

Simulation studies of parallel algorithms to be run on PASM were a major part of this author's earlier research [Kue81, SiK80, SiK81, KuS81]. These studies began with the definition of a target architecture and included studies of custom processors and their instruction sets. Detailed simulation studies

entailed writing a parallel assembly language and assembler, defining how long each custom assembly language instruction would take (in cycles) for the target processors, writing a register-transfer-level simulation program to emulate the processor instructions, and finally developing the application algorithms written in the parallel assembly language. As a result of these studies, it became apparent that the characteristics of the assembly language driven by the image processing applications were no different from those characteristics already provided on commercial microprocessor products. Also by this time, advances in microprocessor technology began to make the speed advantage of the bit-slice approach less clear-cut. This led to studies of standard commercial microprocessors that might be suitable as the processors in PASM PEs. One of these, the Motorola MC68000 [Mot84a], was selected for further study in mid-1981. This microprocessor became the computing engine of the PEs in the PASM prototype and was later adopted for use as the basic unit of computation in the MCs, System Control Unit, and Memory Management System of the prototype.

The high-level design of a PASM prototype system based on MC68000 CPUs was carried out between mid-1981 and late 1982. It included simulation studies and specific design proposals which are described in detail in Part III of this thesis. These studies were originally published in [SiK82] and [KuS82]. In mid-1982, a proposal in which funds for PASM prototype development were included was submitted by Howard Jay Siegel. This proposal was not funded and the author and other members of the PASM group turned their attention to related studies such as the further definition of the functions of the Memory Management System [KuS83], analysis of the PASM operating system [TuS81, TuS82a, TuS82b, TuS84a, TuS84b, TuS85], tradeoff studies of different multistage networks [McA81, SiM81a, AdS82a, AdS84a, AdS84b, SiM81c, McS82a, AdS82b, McS82b, Sie85], definition of new parallel programming languages [CIS83, CIS84, KuS85], study of performance measures [SiS82b, SiS82c, KuS86b], consideration of various PASM implementations [SiS84, MeS85, KuS85, SiK85, KuS85, KuS86a], and studies of a large variety of parallel image processing algorithms [SiS81b, SiD81, Sie81, MuD82, SiS82a, SiS82b, WaS82, SiS82, SiS83, TuA83, RiS83a, RiS83b, KuS84, KuS85, Rij85, KuF85].

Early in 1984, an equipment grant from International Business Machines, Incorporated (IBM) was obtained by Howard Jay Siegel and the *PASM Parallel*

Processing Laboratory was formed. Work on the PASM prototype implementation began in earnest. The remainder of Part I of this thesis details the hardware design tradeoff studies made between 1981 and the present time, considering both the prototype implementation as well as the implementation of a 1024-processor PASM system. Later in the thesis, related studies of parallel programming languages, the prototype operating system, and image processing applications are presented.

CHAPTER 6

PASM DESIGN STUDIES

This chapter details the tradeoff studies that were performed which led to a practical implementation for PASM. Wherever possible, the studies are generalized so that the conclusions apply to all PASM implementations: prototype or 1024-PE PASM. An earlier chapter gave general function descriptions of each of the main components of PASM. Later chapters in Part I of this thesis are devoted to specific details of the PASM prototype and a proposed 1024-PE PASM system.

In Section 1.6.1, a general functional description of each of the PASM system components is given. Section 1.6.2 describes the design of a PE based on a commercial microprocessor. It is shown that the interface logic for the microprocessors necessary for SIMD/MIMD processing is minimal; thus the high cost of a custom LSI or bit-slice processor design can be saved. Some general comments about interconnection network designs for PASM are given in Section 1.6.3. Section 1.6.4 discusses several candidate organizations of the MC. Tradeoff studies of each approach are given. A summary of earlier SIMD algorithm simulation studies and their impact on the design of the MC/PE interface is presented in Section 1.6.5. Section 1.6.6 discusses the functions and requirements of the secondary storage system. Control and communication among the MCs, SCU, and Memory Management System is treated in Section 1.6.7.

1.6.1 Functional Description of System Components

Here, a functional description of the interactions of the MC, PEs, network, and other system components is presented. Later, the implementation options considered for each function will be described. These options include various mixes of hardware and software approaches.

The basic functions of an MC in SIMD and MIMD modes are summarized below. For the following, the key "S" indicates a function needed for SIMD mode, "M" a function needed for MIMD mode, and "B" a function needed to support both modes.

- (S) Fetch SIMD instructions from an MC memory unit and identify them either as control or as PE instructions.
- (S) If an SIMD control instruction is fetched, execute it within the MC.
- (S) If an SIMD PE instruction is fetched, broadcast it to the PEs. If the MC is part of a partition involving multiple MCs, ensure that all of the MCs broadcast the same instruction at the same time.
- (S) Store, decode, manipulate, and broadcast masks specifying which PEs are to be enabled or disabled.
- (S) Receive, store, and manipulate data conditional mask information from the PEs. For multiple-MC partitions, communicate data conditional results local to an MC-group to the other MCs in the partition.
- (M) Execute coordinating programs to manage PE resources or synchronize processes operating on the PEs. Communicate coordination directives to the PEs.
- (B) Allow PEs to request private communication with the MC for the purposes of alerting it to a PE fault or error.
- (B) Receive directives from the SCU indicating jobs to be run, stopped, or killed. Respond to the SCU when jobs are complete or found to be in error.
- (B) Request service from Control Storage.
- (B) Request service from the Memory Management System.
- (B) Detect faulty PEs or faulty program operation that cannot be detected by the PE itself (e.g., an infinite loop in a user program) and take corrective action.

- (B) Initialize and test external devices (e.g., I/O ports). Establish communication with PEs, re-initializing them if necessary.

Similarly, the basic functions of a PE in SIMD and MIMD modes are summarized.

- (S) Fetch SIMD instructions broadcast by the MC and execute them. For data references, read or write the local PE memory unit.
- (M) Fetch instructions and data from the local PE memory unit using a standard fetch-execute cycle.
- (S) Evaluate data conditional masks and communicate the local result to the controlling MC.
- (B) Communicate with other PEs using the established protocols of the interconnection network.
- (S) Detect faulty user program operation (e.g., divide by zero, bus error) and alert the MC.
- (M) Detect faulty user/system program operation (e.g., invalid access to I/O resources) and take corrective action, possibly including alerting the MC.
- (M) Coordinate among multiple processes operating on the PE and manage local PE resources.
- (M) Receive directives from the MC indicating jobs to be run, stopped, or killed. Respond to the MC when jobs are complete or found to be in error.
- (B) Request I/O service from the MC.
- (B) Initialize and test external devices (e.g., I/O ports), interconnection network connections, and the PE-MSU interface.

The interconnection network has the single task of accepting data from the PEs at its N input ports and routing the data to the PEs connected to its N output ports. Path routing is controlled by network routing or destination tags. The network must allow communication in both SIMD and MIMD modes.

The SCU is responsible for the following functions:

- (B) Support program development and interactive use of PASM.

- (B) Determine resource needs of jobs and schedule them by forming partitions made up of one or more MC-groups.
- (B) Direct the MCs to run, stop, or kill a job. Indicate to the user when jobs are complete or found to be in error.
- (B) Allow MCs to request private communication with the SCU for the purposes of alerting it to an MC-group fault or error.
- (B) Request service from Control Storage.
- (B) Request service from the Memory Management System.
- (B) Detect faulty MCs, Memory Management System processors, or I/O subsystems and take corrective action.
- (B) Initialize and test external devices (e.g., I/O ports). Establish communication with the MCs and Memory Management System processors, re-initializing them if necessary.

Control Storage responds to requests for files in secondary storage from the System Control Unit and MCs. The MSUs respond to requests for files in secondary storage from Memory Management System processors. The Memory Management System processors respond to requests from the SCU and MCs. Memory Management System processors are also responsible for establishing communication with the MSUs and testing them.

Ideally, the PASM system hardware should provide all of the functions directly. In reality however, many functions are more easily and more cost-effectively implemented in software and can be considered to be in the realm of operating systems programs (to be considered in Part II of this thesis). The following sections of this chapter will consider hardware/software hybrid implementations that are simple to design, construct, and program.

1.6.2 PASM PE Design

Custom vs. Off-the-shelf Processors

In an SIMD machine, the CU fetches instructions from its memory, executes the control flow instructions (e.g., branches), and broadcasts the data processing instructions to the PE processors. All existing machines designed to operate only in SIMD mode have used custom-designed PE CPUs which receive their instructions from the CU. For these SIMD processors, the complement of operations is limited to the data processing instructions: loading, storing, and arithmetic operations. In an MIMD machine, the CU may coordinate the activities of the PEs, but PEs operate relatively independently of each other. PE CPUs for MIMD mode operations require the full complement of instructions (i.e., require a processor with control as well as data processing instructions). Conventional microprocessors are well-suited to this independent operation.

As discussed in the previous chapter, early PASM writings advocated the use of custom processors for the MC and PEs. This was due to:

- (1) technological reasons -- bit-slice processors of the day were faster and could achieve a wider range of operations;
- (2) a desire for total flexibility -- probably based on the fear that "unknown" characteristics of parallel algorithms would be discovered which would be inefficiently executed on conventional machines;
- (3) efficiency concerns -- duplication of SIMD instruction decoding hardware in the PE CPUs was considered undesirable; and
- (4) precedent -- there were no existing examples of commercial microprocessors being used as control units and PE CPUs in SIMD mode.

Because of recent advances in microprocessor technology, conventional microprocessors now outperform many bit-slice designs and have very large and diverse instruction sets appropriate for a wide variety of applications. Efficiency loss due to duplication of resources (e.g., instruction decoding hardware) is no longer an issue: single-chip hardware solutions are inexpensive and have incorporated architecture concepts such as internal pipelining and

instruction caching which more than make up for the overheads of redundant processing. For these reasons, one of the early goals of this author's research was to determine how to use the same conventional microprocessors effectively in both SIMD and MIMD modes.

Given that PEs are to employ conventional microprocessors, the MC needs to provide the PE CPUs with normal instructions in SIMD mode rather than "control signals" as proposed in [SiM78b]. This seems a logical approach since the PE CPUs will already have instruction decoding for MIMD mode operation; providing it control signals for SIMD mode in addition requires many more pins on the PE CPU integrated circuit package not to mention the large number of lines required between the MC and PEs if control signals were broadcast. Besides, normal instructions can be thought of as a compact encoding of the internal PE CPU control signals.

Bit-Serial vs. Bit-Parallel and RISC vs. CISC

Advocates of bit-serial processors such as STARAN and MPP or architectures in which processors can be combined to form larger word sizes such as TRAC and DCA [KaK78, KaK79, KaK80] claim that those architectures are the best suited for many image processing tasks because the word size can be ideally matched to the problem size. However, bit-parallel processor architectures can often achieve higher arithmetic speeds for multi-bit operands because the bit-parallel data is handled by a single component and need not be moved (carried or shifted) from one processor to another. As clock rates and circuit densities increase, it becomes more and more evident that there are substantial penalties for moving data even from one integrated circuit to another one centimeters away. Of course, the bit-parallel approach may waste precision if its word length is much longer than required by the problem (as it can be due to the predominance of low-precision operations in many image processing applications). Yet many applications require high-precision operations, e.g., floating point operations in certain FFT computations, where bit-serial processing has been found to be uncompetitive with single-chip multi-bit implementations.

Clearly, bit-serial slave processors such as those used in STARAN, MPP, and CLIP-4 would be unsuitable for a general-purpose SIMD/MIMD machine

such as PASM. In order to support MIMD mode, a CPU must be capable of independent operation and support instruction sequencing, control instructions, interrupts, error handling, and the like. The reality is that any CPU capable of independent operation has a significant fraction of its area dedicated to control and sequencing. Therefore, one bit of processing power is too small a "pay-back" for the area dedicated to its control. Current microprocessor technology is driving the size of the internal arithmetic and control structures to 32 bits and higher and internal word sizes smaller than this do not lead to significant savings in chip area. Thus the 16- to 32-bit microprocessor is the smallest reasonable computing element that should be considered for PASM.

On the other end of the spectrum, consider each PASM PE to be composed of a "supercomputer," "main-frame," or "superminicomputer." These approaches are impractical both from a cost and a control standpoint since synchronization of these types of computers for SIMD mode operation would be impossible.

The relative merits of *Reduced Instruction Set Computers (RISCs)* [PaP82, PaS82, FoE84] versus *Complex Instruction Set Computers (CISCs)* have been hotly debated for some time. While the RISC processors are simpler and generally execute individual instructions much faster than CISCs, RISC instructions are generally less powerful than CISC ones; hence, algorithms require more RISC instructions to implement them than CISC ones. It is generally acknowledged that the smaller set of RISC instructions makes compilers and other utility programs easier to construct and maintain. To date, there are no clear-cut advantages to the use of either approach; therefore, the choice of a RISC or CISC processor should matter little to the overall PASM performance. However, the faster clock rates of RISC processors necessitate a faster interconnection network, memory, faster SIMD instruction broadcasting, and faster synchronizing circuits to keep pace. Therefore, a CISC approach may prove to be advantageous for use in PASM, both from a cost and an ease-of-design standpoint.

SIMD Mode Operation

A PE CPU operating in SIMD mode has two specialized needs: the need to fetch and execute instructions broadcast to it from the MC and the need to enable/disable itself. The following discussions address these needs and outline the design tradeoffs involved.

A typical processor bus cycle begins with the CPU placing the current value of its program counter (PC) on the address bus. It also typically outputs "function codes" that indicate what type of reference is desired (read/write), the size of the transfer (one byte, two bytes, etc.), and the logical space and privilege status (data, program, user, supervisor). Shortly thereafter, to allow settling of these bus signals, the CPU asserts an "address valid" signal. Address decoding logic determines which external device or devices (memory, I/O ports, etc.) are to be enabled by the address. The signal actually enabling the device is derived from the address decoding logic gated with the address valid signal. For a read operation, the responding device drives the requested data onto the data bus. For a write operation, the responding device latches in the data appearing on the data bus. In a system with an *asynchronous* bus, the responding device is responsible for replying to the CPU with an acknowledge signal when it has put the data on the data bus (during reading) or has latched the data (during writing). The advantage of an asynchronous bus is that the address lines are driven for as long as necessary; that is, until some device has responded with an acknowledge signal. This allows both fast and slow devices to appear on the same bus. *Synchronous* buses have a fixed time period during which the address lines are valid. Devices must respond within the time period, offering less flexibility in device selection.

Because a conventional PE CPU has no knowledge of the difference between SIMD mode and MIMD mode, it must operate in the same way regardless of from where its instructions come. When a CPU fetches an instruction, it is normally obtained from a memory device. Therefore, the key to providing SIMD instructions is the design of an interface that appears as memory to the PE CPU but which obtains the SIMD instructions from the MC. Note that the value of the PE PC is important in MIMD mode: it selects the next instruction to be executed. In SIMD mode, the PE PC serves only to identify a request for the "next" instruction determined by its MC.

The solution is to build a specialized interface to recognize when the CPU makes program references to a certain range of addresses. If the interface is addressed, it obtains an instruction from off-board and places it on the data bus just as if normal memory were addressed. The interface must respond to a range of addresses because each time an instruction is obtained via the interface (by having a PC value in the range), the PC is incremented in preparation for fetching the next instruction. Thus if the interface responds to addresses in the range X to Y (*the SIMD Instruction Space*), any PC value in that range placed on the address bus would cause the interface to provide the same "next" instruction. When a CPU in MIMD mode wanted to enter SIMD mode, it would "jump to X." If the MC wanted its PEs to enter SIMD mode, it would broadcast a "jump" instruction to some address not in the SIMD Instruction Space. This is how a CPU can dynamically change between SIMD and MIMD mode operation.

The above scheme does have a few minor problems. If the CPU is operating in SIMD mode, eventually its PC will approach and exceed the high end of the SIMD Instruction Space. If it is allowed to exceed it, it will stop obtaining SIMD instructions which is not desirable. Therefore, it is the responsibility of the controlling MC or the SIMD instruction interface to periodically provide "jump to X" instructions to their PEs. Having the MC provide the jump instruction is preferable because it forces all of the PE CPUs to perform it simultaneously. If the jump instruction were generated local to each PE, other PEs would have to wait while the local one jumped to maintain instruction synchronization. Because modern CPU address spaces are quite large (often 2^{24} to 2^{32} bytes), the size of the SIMD Instruction Space can be made large making the overhead of resetting the PE PCs minimal.

An asynchronous bus makes this scheme for obtaining instructions from off-board simple: if no instruction from the MC is currently available, the interface does not assert the bus acknowledge signal and the CPU "waits." On a synchronous bus, some device *must* respond during the bus cycle; otherwise, the results are undefined. Therefore, if no instruction from the MC is available, the interface is made to respond with a "no-operation" instruction which is dutifully executed by the PE CPU.

Given the interface described above, disabling a PE in SIMD mode is simple. An N-bit general mask can be associated with each instruction broadcast by the MC. Bit i of the general mask is broadcast to PE i along with the instruction. The instruction interface always accepts the instruction broadcast by the MC but if the general mask bit indicates that the PE is to be disabled, the instruction is not forwarded to the CPU. In an asynchronous-bus system, the interface simply refuses to provide an acknowledge signal to the CPU; thus the CPU "waits." In a synchronous-bus system, "no-operations" are provided to disabled PE CPUs. An interface that provides a "no-operation" has a slight advantage in that the PEs always complete a bus cycle within a predictable interval and that the PE PCs can be made to remain synchronized. The importance of this will be considered later.

Error Handling

Effective computer designs always provide mechanisms for detecting asynchronous external events (*interrupts*) and for detecting user and system program errors (*exceptions*). Interrupts generally come from I/O devices that are requesting service from the CPU. Exceptions are potential error conditions that the CPU experiences during normal instruction processing. *Internal exceptions* are those that the CPU detects internally. Typically, they include division by zero, attempts to execute privileged instructions, or attempts to execute "instructions" with bit patterns the CPU does not recognize. *External exceptions* are those detected by external hardware and are reported to the CPU. These include access of non-existent or protected memory, attempts to write ROM, and system reset. Interrupts and exceptions cause the CPU to save its current state and to begin processing the code of an interrupt or exception *handler* which performs some action depending on the type of event that occurred. Some examples of handler actions are aborting the current process if it accessed non-existent memory, getting a character from a device signaling an interrupt, or printing a message if the current process divided by zero. Most CPUs allow interrupts to be "masked" to temporarily prevent the CPU from performing interrupt processing. By contrast, exceptions cannot be masked and always cause a handler to be executed.

In MIMD mode, interrupts are unmasked and exception and interrupt handlers are executed local to the PE. This allows the MC to interrupt and communicate with each PE asynchronously for the purpose of scheduling new work on the PE or informing the PE that a disk I/O operation has been completed. If an error occurs in one of the processes associated with an MIMD task that cannot be recovered from locally, the PE should alert the controlling MC so that it can take further action. The protocols for this are operating system-dependent and are discussed further in Part II of this thesis. An example of where this interaction is desirable is illustrated by the following. Consider two processes executing on different PEs interacting in a "producer-consumer" communication model [Sto80]. If the producer process should cause an exception it can be terminated by the PE executing it; however, the consumer process should be terminated as well. Conversely, if the consumer process is in error, the producer process should be terminated.

SIMD error handling is somewhat more involved than that for MIMD mode. Interrupts should not occur in SIMD mode since there are no valid interrupt sources. An MC has explicit control over the instructions a PE CPU executes and thus can communicate whatever it wishes to any set of PEs; therefore, it does not need to interrupt individual PEs. Further, timers and I/O devices are used synchronously; therefore, interrupts are unnecessary for these devices. By contrast, exceptions caused by program errors can and will occur in SIMD mode. When an exception occurs in a PE, it executes the handler instructions stored in its local memory. Therefore, handlers are always executed in MIMD mode. Since the PE with the exception is temporarily fetching and executing instructions locally, other PEs in the SIMD partition must wait because, by definition, SIMD instructions are executed by all enabled PEs at the same time. When the exception handling is complete, the excepting PE can fetch the next SIMD instruction and normal SIMD processing can continue. Exception processing in SIMD mode is rather undesirable since the excepting PE makes the other PEs wait. Fortunately, most exceptions indicate serious errors for which there can be no reasonable recovery; therefore, the exception will result in termination of the current process rather than an extensive "cleanup" of the error condition that would delay other PEs.

In asynchronous bus systems, there is a "bus timeout" mechanism that is triggered if no device responds in a "reasonable" amount of time. Such a mechanism is crucial for detecting the improper use of I/O devices resulting from programming errors. Systems for which the address space is not fully decoded (allowing access to non-existent memory or devices) also rely on the bus timeout mechanism to detect the error. A difficulty with the use of a bus timeout condition in SIMD mode is that there is no "reasonable" amount of time a PE in SIMD mode might have to wait: the MC might be busy attending to some high-priority event and not currently broadcasting instructions or the PE might be disabled for long periods. To solve this problem, the bus timeout mechanism must be disabled for accesses to the SIMD Instruction Space.

Local PE Memory

As described in the general PASM design, each PE is to have two local memory units so that loading/unloading can be overlapped with computations. Early design studies [SiK79] implied that if the CPU is currently accessing memory unit A, it should not be allowed to access memory unit B because it would disrupt the secondary memory transfer. In fact, single-ported memories that were switched between the CPU and the secondary storage buses were proposed. This caused a difficulty for those applications that could not fit in a single memory unit: multiple *frames* of data and/or programs would have to be loaded and processed by alternating between the two memory units. Also, results calculated and stored in one memory unit (e.g., local variables) were not available in the other memory unit and vice-versa. This led to several schemes being proposed that allowed the same local variables to be accessed regardless of which memory unit was being used [SiK79]. In one, a separate local variable memory was proposed. This approach was discarded because the number of local variables could not be estimated: there are none for some applications and for others a complete memory unit might be required to store them. Another approach was to reserve an area (whose size was software selectable) in each memory unit as a local variable area. Both areas would be available to the CPU for local variable storage. The problem here is that new tasks might not be able to be loaded in the other memory unit (due to part of it being reserved)

until the current task completed and canceled the reservation. A third approach is similar to the previous one, but data written to one memory unit would be "written through" to the other memory on a cycle-stealing basis. Here the main objection is complexity.

The use of dual-ported RAM memory units would be an ideal solution because data could be read/written to either memory unit by either the CPU or secondary storage device without interfering with each other. However, dual-ported RAMs are expensive and rather small in capacity. A realistic compromise that allows both devices access to each memory, is cost-effective, and relatively contention-free is a dual arbitrated-access memory. So long as the CPU made its references to one memory unit, and the secondary storage device to the other, there would be no contention. Yet, if the CPU required a data item from the other memory unit, it could obtain it at the cost of an arbitration cycle. If there was a small number of these arbitration cycles, neither device would be significantly affected. In the worst case where both devices are continually referencing the same memory unit, the arbitration circuit provides each alternating access. This scheme has the advantage of not requiring a "reservation" for local variable space nor does it require that a switch be explicitly set to connect the CPU or disk to a certain memory. The arbitration circuit adjusts its operation automatically based on the locality of the memory references.

For the best performance, a parallel data path should be used to connect the secondary storage system (the MSU) to a PE memory. A small area of the PE memory may be reserved as an exchange point for messages to be passed between the MSU and PE. The types of information to be exchanged will be discussed in Part II of this thesis.

The size of memory is highly application-dependent. Nonetheless, some decision must be made that meets the requirements of most algorithms. Even very sophisticated MIMD programs rarely exceed a few hundred kilobytes, but data sets may often achieve this size or larger. Therefore, data is usually the overriding memory-user in the PEs. For applications that require each PE to have a complete copy of the data, the minimum memory requirements are fixed. However, for applications that perform "local" processing of segments of the data in parallel, the per-PE memory requirement is reduced as N increases.

Shared Memory

Private (local) processor memories have typically been employed in parallel processors being designed for image processing applications (e.g., MPP, CLIP-4, ZMOB). This is due to the large percentage of local data references as compared to global references that characterize the low-level processing of images. On the other hand, there are applications and algorithms for which it is desirable for one PE to have access to data in other PEs' memories. An example is an image understanding algorithm that uses "global" knowledge to recognize certain patterns. Operating system programs may also require shared data for synchronization and control purposes.

In SIMD mode, inter-PE communications are inherently structured and synchronized by the single instruction stream program. Consider the communication of data in PASM MIMD mode. To obtain a data item from a remote PE, a transaction request must be generated, encapsulated in a message, and sent to the remote PE. When the remote PE replies, the returned message is decoded and handled. Several recent MIMD machine designs have included both local and shared memory access capabilities. In the BBN Butterfly machine, 24-bit addresses generated by the PE CPUs are monitored by a microprogrammable bit-slice co-processor. Some addresses are determined to be local and are handled by local dynamic memory or I/O ports; others are interpreted as global. Part of each Butterfly PE's memory is shared and appears as one contiguous address space within the aggregate of processors. Therefore eight of the higher-order bits of the global address identify the remote PE number (0 to 255) while lower-order bits indicate the shared memory word in that PE. When global addresses are generated, a message indicating the address and the transaction to be performed is sent through the network to the remote PE holding the shared address. The co-processor in the remote PE steals memory cycles to perform the transaction desired. If the transaction was a read, the data is returned to the initiating PE's co-processor (which remembers that a transaction was pending) which in turn provides the data to the CPU. A similar scheme occurs in RP-3, but the address spaces dedicated to local and shared memory can be set by software providing an all-local, all-shared, or mixed memory access scheme.

PASM can use a hardware scheme similar to that employed in the BBN Butterfly and RP-3 machines or can use a software emulation scheme. In the hardware scheme, a *Network Interface Unit (NIU)* would interpret whether the generated addresses are local or shared. Such a unit may treat a fixed set of addresses as shared or may have internal registers which can be set by the local PE operating system to define the bounds of the local and shared spaces.

One way in which PASM can emulate a shared memory scheme in software is as follows. In each PASM PE's address space, a certain range of addresses will be considered shared; generating an address in this range will cause local address decoding hardware to generate a "bus error exception." The address range can be hard-wired or determined from special registers set by the local PE operating system to define the bounds of the shared address space. During bus error exception processing, the internal state of the CPU is saved on the stack (even if mid-instruction) and the CPU begins to execute an exception handling routine. Upon examination of the stack, the address that caused the bus error can be determined. If it is found to be an address in the shared memory area, it is interpreted as a reference to a remote PE's memory and a message is generated and sent to the appropriate remote PE. When the message arrives there, that PE is interrupted, interprets the message, and performs the transaction. If the transaction was a read, the data is returned to the PE that requested the data. The original PE obtains the data, "patches" it into the stack (thus correcting the bus error due to the "non-existent" local data), and returns from the exception handler. Finally, the bus cycle is re-run by re-starting it or continuing it from mid-instruction (depending on the conventions and capabilities of the specific CPU being used).

There is a substantial performance penalty for the handling of a shared memory reference in software. Nonetheless, this scheme offers as much flexibility in choosing the sizes of the shared data areas as does RP-3 and does so without the cost of an NIU. The time needed for a shared memory reference is lower for a machine having a bidirectional network as compared to a unidirectional one. Thus even if the machine has a PE-to-PE configuration, (e.g., PASM), a bidirectional network is useful because it saves the time involved in establishing a network path in the reverse direction. Requested data can be returned along the bidirectional link established by the incoming request.

Even if an NIU is used, simultaneous access of a single memory location by several processors leads to serialization. If a multistage circuit-switched network connects the PE NIUs, only one of the competing NIUs gains access to the remote NIU at a time. If a packet-switched network is used, the remote NIU receives and processes the incoming request packets serially. To prevent the serialization, a *combining* network can be used [GoG83]. In such a network, each (packet-switched) interchange box examines packets at its two input queues and detects if they refer to the same memory location. If they do, the two requests are combined and a single composite request is sent to an interchange box in the next stage. This ensures that even if all processors in the system simultaneously want the value of shared address X, only one request for X appears at the NIU associated with the memory containing X. When the value of address X is fetched and returned along the reverse bidirectional path, each interchange box along the path that combined two requests for X duplicates the data so that it is returned along both requesting paths simultaneously. Thus all requesting PEs receive the data at the same time (neglecting packet delays).

A simultaneous request by many processors for the same address is not necessarily a rare occurrence. As an example, semaphores [Dij68] for controlling access to a critical resource may be tested by many processors simultaneously. A combining network such as the one implemented in RP3 has interchange boxes that can combine primitives based on Fetch-and-Add [GoG83]. Fetch-and-Add primitives can perform read, write, or read-modify-write operations on shared memory. The read-modify-write operation is used to implement counting semaphores; therefore, synchronization operations that rely on semaphore counts can be done in RP3 without serializing access to the semaphore. This dramatically improves the performance of operating systems programs, as will be described in Part II of this thesis.

Memory Management and Virtual Memory

Memory management is a generic term for a number of techniques that can aid in higher memory utilization, program error detection, and enhanced memory fault-tolerance. One technique is the use of a memory divided into mapped pages [HwB84]. Each logical memory address that the CPU generates is translated by getting a corresponding physical address from a page map. Mapping eliminates the fragmentation problem; that is, the inability to allocate a contiguous memory area large enough for a program or data segment. Since the map can be changed under operating system control, programs and data can be relocated to any free memory page, increasing memory utilization. Memory errors due to bad hardware can also be avoided by mapping around them. Also, pages containing program code can be shared by multiple processes since many logical addresses can be mapped to the same physical one.

If protection bits are associated with each memory page, these may be set by the operating system to indicate the access rights of processes using each memory area. Per-page protection bits prohibit operating system data from being read or modified by user programs and also protect users from each other. Without protection, a PASM machine is completely vulnerable to "crashes" or "subterfuge" caused by careless or malicious users. In addition, lack of memory protection allows errors in one user's program to potentially affect the results of other user's programs, making debugging impossible.

A protection scheme necessitates a CPU architecture with at least two levels of privilege: an unprivileged "user" state and one or more privileged "supervisor" states. The current privilege level of the CPU is made available during each bus cycle so that it can be checked by external hardware. Certain resources, notably I/O devices, page maps, and per-page protection bits are accessible only while the CPU is operating in a privileged state (executing the operating system). Users can access I/O devices only through a set of pre-defined *system calls* that cause the CPU to temporarily enter the supervisor state. Only through the use of these mechanisms can the operation of PASM be guaranteed to be correct and orderly.

Virtual memory capability is a relatively new feature of some microprocessors. A virtual memory system [Den70] allows portions of a program's code or data to be kept outside a CPU's primary memory while not being referenced.

This allows programs and data files much larger than the system's primary memory to be run or operated upon by the CPU. The virtual memory system manages the address space of the CPU by detecting when a program or data reference is *non-resident* and initiating exception processing to bring the missing program or data area into primary memory. If necessary, data in primary memory is migrated out to secondary storage to make room for the missing segment. For efficiency, the data is moved in and out of primary memory in page-sized chunks (hundreds or thousands of bytes) rather than word-by-word.

A *fault* is said to occur when the virtual memory system detects a reference to a non-resident data item. The *fetch policy* determines what pages are made resident when a fault occurs. *Demand fetching* is the process described above that causes page-sized program or data areas to be migrated into primary memory as they are needed. There are other fetching policies that attempt to anticipate future references and prefetch the pages. Some of these policies are described in [HwB84]. The *replacement policy* determines what pages are written back to secondary memory when newly-fetched pages are to be made resident. Two common policies are *least-recently-used* and *first-in-first-out* [HwB84].

Consider the use of a virtual memory scheme in the PASM PEs. In SIMD mode, PEs must remain instruction-synchronized. If virtual memory were available in SIMD mode and one out of N PEs faulted, that PE would have to resolve the bus fault, wait for secondary storage to retrieve the missing page, and re-run the bus cycle. This would cause the other N-1 PEs to wait as well. In the worst case, each synchronized memory reference would cause a fault. The chance of a fault occurring somewhere in the system increases as the number of PEs increases. Therefore, the use of paging (due to small primary memories, for example) is not particularly desirable for efficient SIMD processing. In MIMD mode, the situation is less critical because one PE processing a fault does not delay others directly. However, due to many CPUs sharing each secondary storage device, paging may overtax the MSUs.

Because a page fault generally involves the initiation of a secondary storage device access and since there may be a significant delay before the secondary storage request is satisfied, a CPU in a conventional serial computer system frequently switches context to another run-able process when a fault

occurs. In SIMD mode, such a context switch would have to be orchestrated by the MC(s) and would have to be done by all PEs simultaneously. The current states of all participating MCs and PEs would be saved and the state of the suspended process re-loaded. Depending on the amount of state information to be saved and the latency of the secondary storage device, context switching may or may not be practical in SIMD mode. On the other hand, context switching in MIMD mode is desirable since it increases processor utilization.

One desirable attribute of a paging system is that over time, pages of data that have been modified are automatically migrated back to secondary storage under the direction of the replacement policy. This tends to equalize secondary storage usage over the course of program execution and often leaves only a small portion of the pages to be saved in secondary storage at the end of execution. In contrast, systems without paging concentrate disk activity before and after program execution.

The optimal page size for a PE memory is dependent on the application and the characteristics of the secondary storage devices, but in general it can be said that the page size should be larger than that used for stand-alone computer systems. The expected large data sets and the sharing of MSUs among PEs implies that the overhead of each secondary storage operation should be kept as low as possible. Large page sizes also increase the expected time between faults so long as there is a reasonable degree of program and data locality [Den70]. This in turn reduces the number of context switches. The implication is that the PE primary memories should be as large as is practical and affordable to decrease the number of accesses to secondary memory needed during program execution.

Instruction and Data Caches

A *cache* is another form of virtual memory scheme in which an ultra-high-speed memory holds program or data words that are very actively used [HwB84]. An *instruction cache* holds "blocks" (usually a small number of words) of program instructions that have most recently been fetched from primary memory. Instructions are cached in the hope that they will be used

repetitively as would occur if the instruction was a part of a program loop. A cache *hit* occurs when the CPU generates a request for an instruction already in the cache. A cache *miss* occurs when the instruction is not already in the cache and must be retrieved from primary memory. On cache misses, instructions in the cache are discarded according to a cache replacement policy and the needed instructions are fetched according to a cache fetch policy.

Data caches are similar, but hold the addresses and values of data items recently referenced. A *non-homogeneous* cache is one in which both instructions and data occupy the same cache.

Since data is both read/write, while instructions are read-only, data caches can suffer from a problem known as *cache coherency* in which the cached value for a given address may be different than the value in the corresponding primary memory address. The *primary memory update policy* determines how data values that have been modified in-cache are migrated back to the primary memory. This is a significant problem if the memory is shared among processors: if one processor modifies a locally-cached data item while another processor reads the corresponding data item from the shared primary memory, the second processor has received erroneous data. Even if PASM does not have shared memory, there may be a cache coherence problem between a PE CPU with a cache and an NIU or a *Direct Memory Access (DMA)* unit. Many solutions to the cache coherence problem have been proposed; a survey of approaches appears in [HwB84].

Consider the use of instruction and data caches in PASM PEs. In SIMD mode, use of an instruction cache is nonsensical because by definition, SIMD instructions are fetched from the MC rather than locally. If an instruction cache happened to be enabled, each SIMD instruction access would result in a cache miss. In the unlikely event that the SIMD instruction address range was smaller than the instruction cache, instructions would eventually be (erroneously) fetched from the cache rather than from the SIMD instruction interface. For these reasons, an instruction cache, if present, should be disabled during SIMD mode processing. Instruction caches for MIMD mode operation have no undesirable side effects.

Data caches are desirable in PASM PEs for both SIMD and MIMD mode operation. Usually the speed of the cache is only a few times faster than the

speed of the primary memory; therefore, there is little "penalty" incurred by cache misses. In the worst case for SIMD mode, some PE will have a cache miss on each reference rendering the use of the cache ineffective. However, such behavior would not be expected since most practical programs access at least a small set of locations frequently enough to keep them in cache, resulting in some performance improvement. Cache misses do not directly penalize other PEs in MIMD mode; therefore data caches can be used effectively so long as the cache coherence problem is addressed.

Other Specialized PE Functions

The need for a PE CPU to determine its physical number is required in both SIMD and MIMD mode when constructing routing or destination tags for interconnection network operations. As an example, consider calculating the destination address for a network transfer that is specified as a "distance" in the program. In order to communicate with the PE that is "+1" away, the PE number is loaded, added to one, and used to set the network. The PE CPU determines its physical number using one of several mechanisms. One possibility is that the number is burned into an EPROM. This is the simplest solution but replacement of the PE requires that the correct EPROM be inserted. Another approach would be to have the settings of bit-switches readable through an I/O port on the CPU board. The switch positions would be set if replacement of the PE became necessary. A similar solution would be to electrically encode the PE number on the pins of a connector readable through an I/O port. This would eliminate problems resulting from incorrect manual settings of switches since a generic PE board inserted in a certain "slot" in the system would always get the correct PE number. Yet another solution is to write operating system software for the MCs that distributes numbers to each PE at boot-up time.

MC/PE communication is necessary in both SIMD and MIMD modes. In SIMD mode, PEs that have encountered some error condition (e.g., bus error, divide by zero) report it to their MC using an MC/PE communication port. For MIMD mode, PEs use the link to request service by secondary storage devices, to report completion of processes they have executed, to obtain

information about where other processes they wish to communicate with are located, and to gain access rights to shared data structures such as semaphores. Because the traffic on this link is relatively infrequent, a bidirectional serial link is probably sufficient.

As described in an earlier section on masking schemes, the PEs can collectively report their status in a "data conditional mask." Each PE contributes one bit to this mask. Therefore, the design requires that a 1-bit port be provided to which the PE writes the status information requested by the MC. The PE status would typically a condition bit of its internal status register, e.g., "zero," "carry," "overflow."

Direct Memory Access

DMA capability is useful for further enhancing the capabilities of a PE. A DMA transfer is arranged by a CPU which programs the DMA controller to perform some memory-to-memory copy. Since I/O devices are simply specially-responding memory locations, the DMA controller may also perform I/O-to-memory, I/O-to-I/O, and memory-to-I/O device transfers. Typically, a starting address and a byte count is written by the CPU to internal DMA controller registers, followed by a "start" command. Once started, the DMA controller arbitrates with the CPU to obtain bus cycles. Depending on the CPU bus mastership protocol, bus cycles may have to be arbitrated by the DMA unit on a cycle-by-cycle basis or may be taken continuously once mastership is gained until the DMA transfer is complete. The DMA controller may also be programmed to relinquish bus mastership every so often back to the CPU (*cycle stealing mode*) or may be programmed to be *greedy*.

When a bus cycle is granted by the CPU, the DMA controller generates a bus read to obtain a data item. On the next free bus cycle, a bus write is generated. If necessary, additional requests for bus cycles are made by the DMA controller until the transfer is complete. The CPU is generally interrupted by the controller automatically upon completion of the DMA request to inform it that the DMA controller is "free."

DMA capability is desirable because the CPU does not have to explicitly fetch and execute instructions to move data. This is especially important if the

CPU lacks an instruction cache and thus would have to compete with the DMA controller for bus cycles. Another advantage of DMA is that the CPU can return to normal processing after programming and starting the DMA unit. Obviously the DMA transfer proceeds fastest if it can get every available bus cycle. However, this would "lock out" bus access by the CPU (as would programming the DMA controller in "greedy" mode). Therefore instruction and data caches are useful in tandem with DMA transfers since they reduce the number of bus cycles needed by the CPU. Additional desirable capabilities of DMA controllers will be discussed in Part III of this thesis.

1.6.3 PASM Interconnection Network and Interface

As described earlier, a PE controls the network using routing or destination tags. After calculating the tag, the PE writes it to an external *Routing Control Register (RCR)* which instructs the network to set switches to make a connection with the destination address. Data transmissions occur through two external *Data Transfer Registers (DTRs)* [SiS81b]. One register is connected to the network input (*DTRin*), and the other to the network output (*DTRout*). Data to be transmitted is written to the *DTRin* register and incoming messages are received and read at the *DTRout* register. In SIMD mode, all PEs in a partition do the network setting and transmitting operations at the same time; PEs use the network asynchronously in MIMD mode.

Because the interconnection network interface is made up of I/O devices, interprocessor communication is done only through use of pre-defined operating system calls. The system calls ensure that correct communication protocols are maintained; reliance on user code to control the interface could result in the hardware being left in unknown or potentially damaging states. The system calls also enforce the mapping of logical to physical processor numbers and the partitioning of the network. Descriptions of a number of possible low-level interprocessor communication protocols follow.

Use of the interconnection network in SIMD programs results in all of the PEs entering MIMD mode for the duration of the transfer. This is because system calls are always executed in MIMD mode. Therefore, the following protocol discussions assume programming and use of the network in MIMD mode.

To begin, assume that the interface is controlling a circuit-switched multistage Cube-type network with n stages and that destination and broadcast tags are being used. This implies that the RCR will be $3n$ bits wide, n bits for the destination address, n bits for the broadcast information, and n bits for the source address. The source address is not used in establishing the path but is required by the destination PE. From the point of view of a source PE, an interconnection network transfer will consist of three steps: establishment of the path, transfer of the data, and dropping of the path. Correspondingly, the destination PE(s) will accept the path by recording the number of the source PE, accept, buffer, and acknowledge incoming data, and acknowledge the termination of the transfer.

The protocol requires an end-to-end interlocked positive acknowledgement handshake between the network interfaces of the source and destination PEs. This means that in the quiescent state, the destination interface asserts a "ready-for-data" signal to the source and the source interface negates a "data-available" signal to the destination. The source sends data only if "ready-for-data" is asserted and the destination takes data only if the "data-available" signal is asserted. The handshake begins with the source placing data in the DTRin which causes "data-available" to be asserted. When the destination sees this, it latches the data into DTRout and replies by negating "ready-for-data." When the source sees this, it knows the destination has received and latched the data. The source responds by negating "data-available" and signals the empty condition of DTRin by interrupting the source PE or providing a status flag that indicates that DTRin is "empty." Now it is up to the destination PE to read DTRout. The presence of data in DTRout is signaled by interrupting the destination PE or by a status flag that indicates that DTRout is "full." When DTRout is read, it immediately responds by asserting "ready-for-data" back to the source, thus completing the handshake.

For SIMD program transfers, a pre-determined number of data items are expected; therefore, the status flags of the DTRs can be polled by the source and destination PEs. However, in MIMD mode, data can arrive at any time; therefore, the DTRs should be set to interrupt the CPUs to indicate their need for service.

A network transfer begins with the source writing tag information to the RCR. The source processor number may be explicitly written by the source PE CPU to the RCR or may be provided automatically by special hardware. The tag does not begin to propagate through the network stages until a special "set-path" signal is asserted by the CPU. The "set-path" signal also enables a watchdog timer that is set to expire and cause exception processing if errors in setting the network path occur. As the tag propagates through the network stages, the switches in stage i interpret bit i of the destination and broadcast tags. If a conflict occurs at any stage, the tag is blocked at the input to that stage and is not propagated further. If the tag encounters no blockages, it reaches the other end of the network. Here, the destination PE number is checked against the destination tag using the expression $(P \text{ OR } B) \text{ XOR } (D \text{ OR } B)$. If the result is zero, the network routing was successful and the source part of the tag is latched and recorded by the destination PE. The destination PE acknowledges receipt by a handshake which propagates back to the source and indicates completion of the path setup phase. The return handshake also disables the watchdog timer.

If the expression given above is non-zero, the routing was incorrect. In this case, the destination PE ignores the routing by not acknowledging it. Eventually, the watchdog timer will expire and cause the source to initiate exception processing. The source PE may wish to try the path routing again a number of times before giving up and alerting its controlling MC.

Blocked paths in the network can also cause the watchdog timer to expire. For the greatest flexibility, the watchdog timer value should be software-selectable. Small timer values (i.e., just long enough to set the network assuming no conflicts occur) can be used to detect conflicts and to drop the path if it cannot be immediately established. Large timer values (i.e., essentially infinite) can be used to greedily hold the path gained so far. Intermediate timer values allow holding of a path for a time even if blockages were encountered, but allow true errors to be detected within a reasonable time period. Here again the source PE may wish to try a particular path routing a number of times before giving up and alerting its controlling MC.

Assuming that the network routing was established correctly, data transmission can begin. Each data item placed in DTRin causes a watchdog

timer to be enabled. The timer value here is made just large enough to accommodate the network propagation time of the data and the return acknowledgement plus the time needed by the destination PE to respond to an interrupt and retrieve the data from DTRout. If the timer expires, it indicates that a data transmission error or a fault in the destination PE has occurred. Again, the source PE may wish to try to re-transmit the data or even drop and re-establish the path a number of times before concluding that the network is truly faulty.

The final step is the dropping of the established path. This is done by negating the "set-path" signal. The signal propagates from stage to stage, freeing the established path and allowing formerly-blocked paths to be formed. When the negated signal is received at the destination, the polling routine (if any) is canceled, any buffer cleanup operations are performed, and control is returned to the caller of the system routine that effected the transfer. Finally, the negated "set-path" signal is reflected back to the source where similar cleanup operations are performed (such as freeing buffers), followed by a return to the caller.

An alternative to the explicit assertion and negation of the "set-path" signal would be to automatically start the path establishment phase when the RCR was written with a valid tag and to drop the path when the RCR was written with an invalid (reserved) tag. Still another possibility would be the use of a special "COUNT" register in the network interface. A write to COUNT would establish the path, transfer the first COUNT words to be written to DTRin, and automatically drop the path. However, this last scheme may not be as flexible when errors occur and retransmission is attempted because the COUNT register would have to be adjusted without starting a new transfer operation.

Note that a protocol that does not terminate a transfer by dropping the established path could lead to deadlock even in SIMD mode. The reason is that if PEs become disabled, they no longer participate in transfer operations; thus they may be holding links required by active PEs. Therefore, paths must always be dropped just before a system call that performed a transfer returns control to the user program. This may be a barrier to optimization for some SIMD programs, for example, those for which the same interconnection

function is used throughout the program or those for which PEs are always enabled.

For SIMD mode data transfers, the destination PE simply buffers the incoming data for the user program that made the transfer system call. No additional information needs to be associated with the incoming data because the routing and amount of data transferred is implicit in the SIMD algorithm. On the other hand, MIMD mode transfers require that extra information precede the actual data being transmitted. This information must include the identifiers of the process sending the data and the process that is to receive it on the destination PE. It is conceivable that multiple processes will be running on each PE in MIMD mode; therefore, the correct recipient must be identified. The destination PE will buffer the incoming data on a per-recipient-process basis, tagging each incoming message with the process identifier of the sender.

Consider the (unlikely) situation where the network propagation delays are much longer than the time needed to read and write the DTRs. In this case, it may be desired to overlap processing with network transfers to maximize throughput. Unfortunately, a safe protocol for doing this is in SIMD mode does not exist: separate system calls for putting items into the network and taking items out would leave open the possibility that such calls would be mismatched. Nor can a non-synchronous transfer (i.e., one in which the system call returns before the operation is completed) be allowed because of the prohibition of interrupts in SIMD mode. Of course in MIMD mode, the normal system calls allow unmatched and non-synchronous reads and writes of inter-process (and inter-processor) data. Here, interrupt-driven I/O can be used to perform data transfers "in the background" at the maximum rate the interconnection network allows.

If the network DTRs are also DMA-compatible, end-to-end DMA transfers from the source PE's memory to the destination PE's memory through the interconnection network are possible. Moreover, this is only possible with a circuit-switched network.

Now consider the low-level protocols associated with the use of a packet-switched multistage Cube-type network. The first difficulty is that all messages must be split into packets and have a tag prefixed to them. Since the tag consists of $3n$ bits, the packet size *must* be at least this large and should

actually be chosen much larger to minimize the overhead due to sending it. Unfortunately, a packet size chosen to be too large may not be efficient either, especially if short messages are the rule. Large packet sizes also imply that the switching elements are complex due to their internal storage requirements and that end-to-end response time will be degraded since complete packets are completely absorbed at each switch before being passed on. Further, a tag contributes to the traffic in the network; conceptually then, a packet-switched network has to have a higher bandwidth to accommodate the same data traffic as a circuit-switched network.

Another problem with packet-switched networks is that there is no implicit end-to-end acknowledgement. If errors occur between switches mid-network, a source PE is not aware of the problems and may continue to submit packets into the network. Only through an explicit positive-acknowledgement-with-retransmit (PAR) protocol and packet sequence fields [Tan81a] can the source PE ensure that the destination PE(s) receive the data. Both the acknowledgement traffic and sequence field bits contribute to the additional bandwidth needed in a packet-switched network. Furthermore, some device has to extract data from packets, examine sequence fields, process acknowledgements, and handle retransmissions. If the PEs themselves are used to enforce these protocols, it keeps them from doing more productive operations.

The rather stringent processing demands of a PAR protocol indicates that an NIU would be of great use in a packet-switched environment. Such an NIU would be programmed by a PE in much the same way as a DMA controller: the starting address, count, and tag would be written to the controller's internal registers. The NIU would autonomously fetch the data, establish a path, perform the transfers according to the selected protocol, drop the path, and interrupt the source PE when done. At the destination NIU, packets would be received and buffered and acknowledgement packets would be created and sent. Although in circuit-switched mode the NIU is conceptually much simpler, it may still be of some use in insulating the PEs from having to handle error conditions explicitly.

Finally, consider the use of networks in which the performance is degraded due to faults. For example, assume the use of the 1-fault-tolerant Extra Stage

Cube' network with a single fault in an internal link or switch. Although all source-destination pairs can communicate, they can no longer do so simultaneously. If a simultaneous transfer is requested in an SIMD program, the system call controlling the network must determine how to accomplish the transfer using multiple steps. Based on the tag and a knowledge of the location of the error, a source PE can determine if the path is affected by the error [DaH85a, DaH85b]. If so, the tag can be transformed such that the alternate routing is taken.

Since each PE calculates and transforms tags independently, there is no global knowledge about what transformed paths will conflict. Therefore when tags are submitted by the PEs to the network, one or more paths will be blocked and the corresponding source PEs will be forced to wait to perform their transfers until after all of the other PEs have finished. This further illustrates the necessity for performing all interconnection network transfers in MIMD mode. When network faults are detected, the value of the path establishment watchdog timer and data transmission watchdog timer must be adjusted to accommodate this "serialization" of the inter-PE transfer.

Network Summary

Some of the tradeoffs between circuit- and packet-switched interconnection networks have been enumerated. The choice of the network type will depend upon factors such as task characteristics, the state-of-the-art at implementation time, and budgetary constraints. Furthermore, the preceding discussion is based on the construction of the interconnection network using off-the-shelf discrete logic. Of all of the components in PASM, the network seems most appropriate for a custom LSI implementation because of its high degree of regularity.

1.6.4 PASM MC Design

There are a variety of design and implementation options available for the PASM MCs. An MC actively participates in both SIMD and MIMD processing, but does so for the two modes of parallelism in very different roles. In SIMD

mode; the MC acts as the control unit: fetching SIMD instructions, executing the scalar and control flow instructions (e.g., initialization, loop control, branching), and broadcasting the data processing instructions to the PEs. It must do all of these functions at a rate sufficient to keep the PEs busy. In MIMD mode, the MC is no less active: it aids in the assignment of MIMD processes to processors, fields I/O requests from the PEs, acts as a clearing-house for information about where (which PE) a process resides, handles transactions on shared data objects such as semaphores [Dij68], and communicates status information to the SCU. Because of these diverse activities, the PASM MCs need to have both general-purpose computing ability as well as a number of specialized components to help them provide the necessary services. In this section, much attention will be given to the organization of the MC and its interface with the PEs.

As discussed previously, the use of custom processors and specialized MC hardware was advocated in early PASM writings. Later, as the scope of the capabilities required for PASM MCs became known, it became clear that an all-custom approach, while capable of high performance, would be difficult to design and construct. MC organizations that allowed use of commercial microprocessors and other pre-developed components could be assembled at lower cost and require less design time.

In the following discussions, several functional elements are referred to: the *MC CPU* has a conventional instruction set and performs control operations in SIMD and MIMD mode; *MC CPU Memory* holds instructions executed by the MC CPU; the *Fetch and Broadcast Unit (FBU)* fetches SIMD instructions, decodes them, passes the control instructions to the MC CPU, and broadcasts the PE instructions to the PEs; and *FBU Memory* holds SIMD instructions. An instruction to be broadcast to the PEs is placed in the *Instruction Broadcast Register*. The current N/Q-bit general mask vector that an MC uses to specify the enabled status of its PEs is held in the *Enable Signal Register*. PE address masks are decoded into an N/Q-bit general mask vector by the *PE Address Mask Decoder*. Data Conditional Masks are held in the *Data Conditional Mask Register*. Since these are functional elements, one or more of the functions may be implemented in a single hardware component or even by software.

Several different organizations of an MC are presented. They differ in the amount of control that the MC CPU and FBU have and schema for the placement of SIMD and MIMD instructions in memories.

A commercial CPU alone cannot be used as an MC unless it is significantly faster than the PEs: it must "feed" not only itself with instructions, but provide the PEs with a constant stream of instructions to avoid "starving" them. Since the PEs are already assumed to be "fast," this organization option will not be explored.

Combined MC CPU/FBU Organization

First consider an MC consisting of two principal hardware elements: a combined MC CPU/FBU and a combined MC CPU/FBU Memory (Figure 1.6.1). The MC and PE instructions of SIMD programs are stored together in the single memory. MC control programs for MIMD mode operation are also stored here.

A typical SIMD fetch-execute cycle for this organization is as follows. As in a conventional processor, the FBU fetches instructions by placing the current value of its program counter (PC) on the address bus and waiting until the memory asserts an acknowledgement (indicating that the required instruction word is on the data bus). It then examines the instruction word to determine if it is an MC or PE instruction. If an MC instruction, it is passed to the internal MC CPU to be executed; otherwise, the instruction is written to the Instruction Broadcast Register and is made available to be fetched by the PEs enabled by the Enable Signal Register.

As stated earlier, an SIMD instruction may be one of two types: MC or PE. MC instructions are those that cause some action to be taken by the MC CPU: initialization, loop counting, etc. PE instructions are those that are to be broadcast to the PEs. Assuming the PEs are centered around commercial microprocessors as advocated in the previous section, PE instructions can be passed to the PEs as conventional instructions rather than as PE control signals. This differs from the way most existing SIMD-only machines operate: their control units decode instructions and broadcast control signals (e.g., Illiac IV, MPP, PEPE). In those machines, the control unit examine (decodes) all

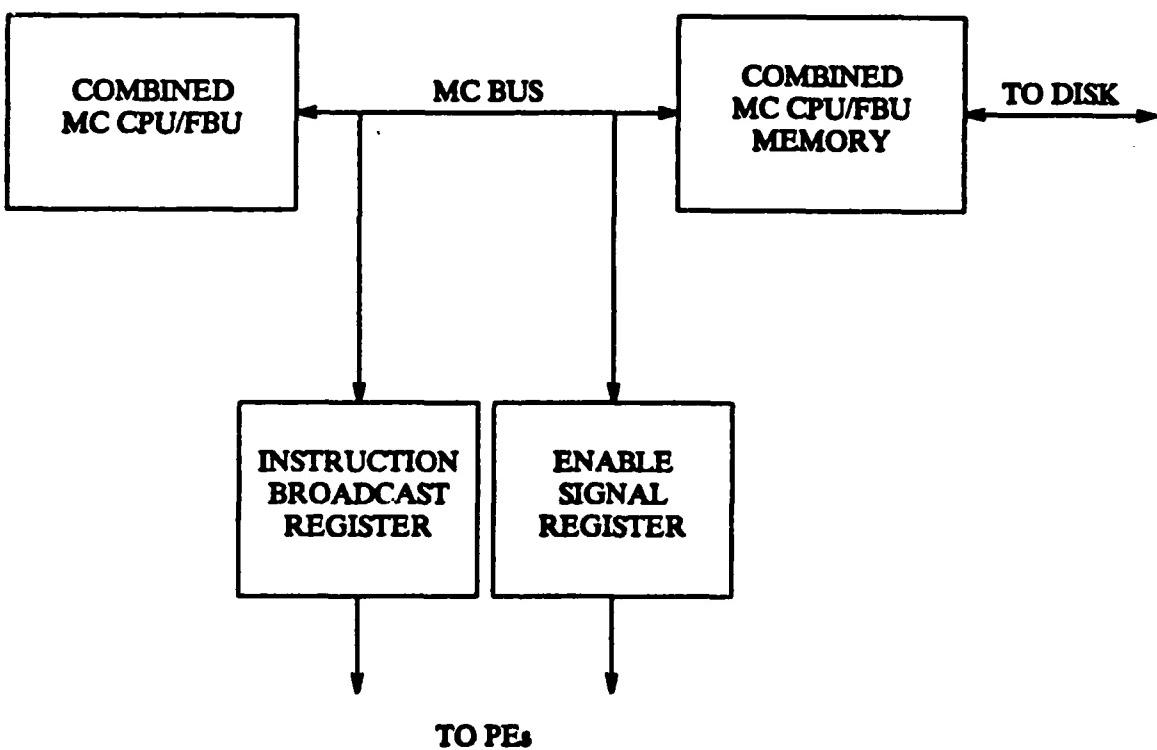


Figure 1.6.1. MC organization consisting of a combined MC CPU/FBU and combined MC CPU/FBU Memory.

instructions to determine their type. Having the control unit perform all of the decoding functions saves the time of re-decoding instructions in the PEs and eliminates duplication of the decoding hardware in each PE.

MC and PE instructions need to be distinguished by their bit patterns. If PE instructions are destined for a conventional microprocessor, they are usually sequences of 8- or 16-bit "opcodes" and "operands." Since opcodes and operands are indistinguishable, some way of identifying MC and PE instructions to the FBU needs to be developed. Perhaps the easiest method is to tag each instruction word with a bit that indicates what type of instruction it is. Unfortunately, this results in the memory being a non-standard width, i.e., 9 or 17 bits. Another possibility is to have the assembler (the program that converts assembly language into object code) insert a reserved opcode in the instruction stream that indicates the subsequent instructions are of "type MC" or "type PE." The FBU would assume that type until another reserved opcode arrived.

The approach just described may suffer from insufficient bandwidth of the MC Bus. Note that the MC Bus is used twice for each PE instruction: once to fetch it from memory and once to place it in the Instruction Broadcast Register. A somewhat better approach would be to send the PE instruction to the Instruction Broadcast Register using a separate data path (see Figure 1.6.2). This would reduce traffic on the bus and would allow some instruction pipelining; that is, while a PE instruction is being broadcast, the next instruction could be fetched from the memory and decoded. The main drawback to this approach is that it increases the number of separate signals that the MC CPU/FBU must produce.

Even if the separate Instruction Broadcast Bus is implemented, the MC Bus may still suffer from insufficient bandwidth. The reason is that MC and PE instructions as well as MC data are fetched using this bus. Therefore, the MC Bus bandwidth would have to be significantly higher than that of the PE buses to keep the PEs satisfied. Some possible ways of reducing the MC Bus usage are the use of instruction and/or data caches or the use of an entirely separate bus for MC CPU data accesses.

In MIMD mode, the MC CPU/FBU combination operates as a conventional processor fetching instructions and data from the memory. The

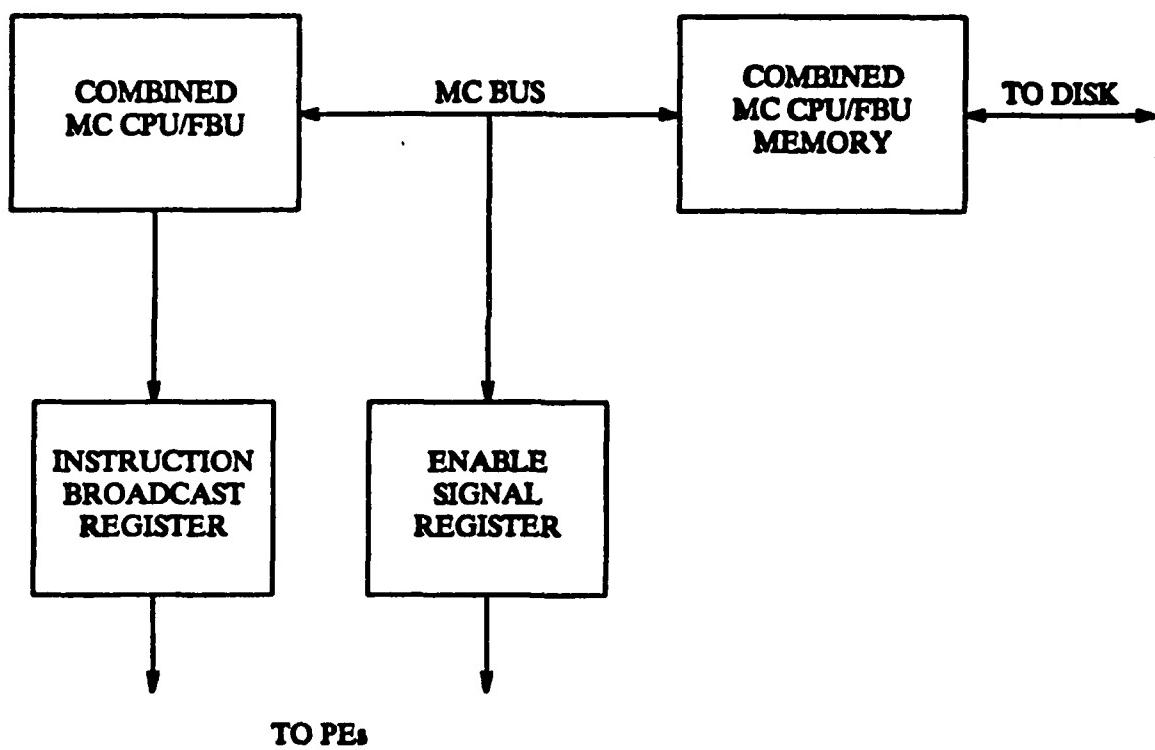


Figure 1.6.2. MC organization consisting of a combined MC CPU/FBU and combined MC CPU/FBU Memory. A separate Instruction Broadcast Bus reduces traffic on the MC Bus.

Instruction Broadcast Bus and the registers associated with PE masking operations are inactive during MIMD mode. During MIMD mode, the MC Bus should have adequate bandwidth available since it is not being used to fetch SIMD instructions.

Since the combined MC CPU/FBU incorporates the non-standard features of instruction decoding and the "separation" of the MC and PE instruction streams, it is likely that an all-custom design would need to be used. This forces the designer to "re-invent the wheel" so far as the MC CPU is concerned. To opt for a very simple control unit (e.g., STARAN, MPP) would sacrifice performance and flexibility needed in MIMD mode. On the other hand, a more complex control unit would require a custom LSI chip design or barring that, many "boardfulls" of components. Clearly, separating the MC CPU from the FBU and using an existing commercial product for the MC CPU considerably reduces the burden on the designer.

Master FBU - Slave MC CPU Organization

Now consider an organization in Figure 1.6.3 in which an FBU "master" fetches SIMD instructions from FBU Memory and determines their type. MC CPU instructions are passed to the "slave" MC CPU and PE instructions are placed on the Instruction Broadcast Bus. Data for the MC CPU (e.g., loop counters) is stored in the MC CPU Memory. In MIMD mode, the MC CPU fetches instructions and data from the MC CPU Memory and the FBU is unused.

In this scheme, the FBU has an internal *FBU Program Counter (FPC)* which gives the address of the next SIMD instruction to be fetched. The MC CPU acts as a "slave" processor: it is "fed" instructions by the FBU in much the same way as the PE is fed. It requests an instruction by placing its PC on the address bus; it receives an instruction when the FBU places one on the MC CPU Bus and asserts the acknowledge signal. The actual PC value the MC CPU generates is irrelevant; it serves only to identify that the MC CPU is ready for the next instruction. When the MC CPU performs instructions, it reads/writes any data items from/to its local MC CPU Memory. For this scheme, the MC CPU Memory contains only data and controlling programs for

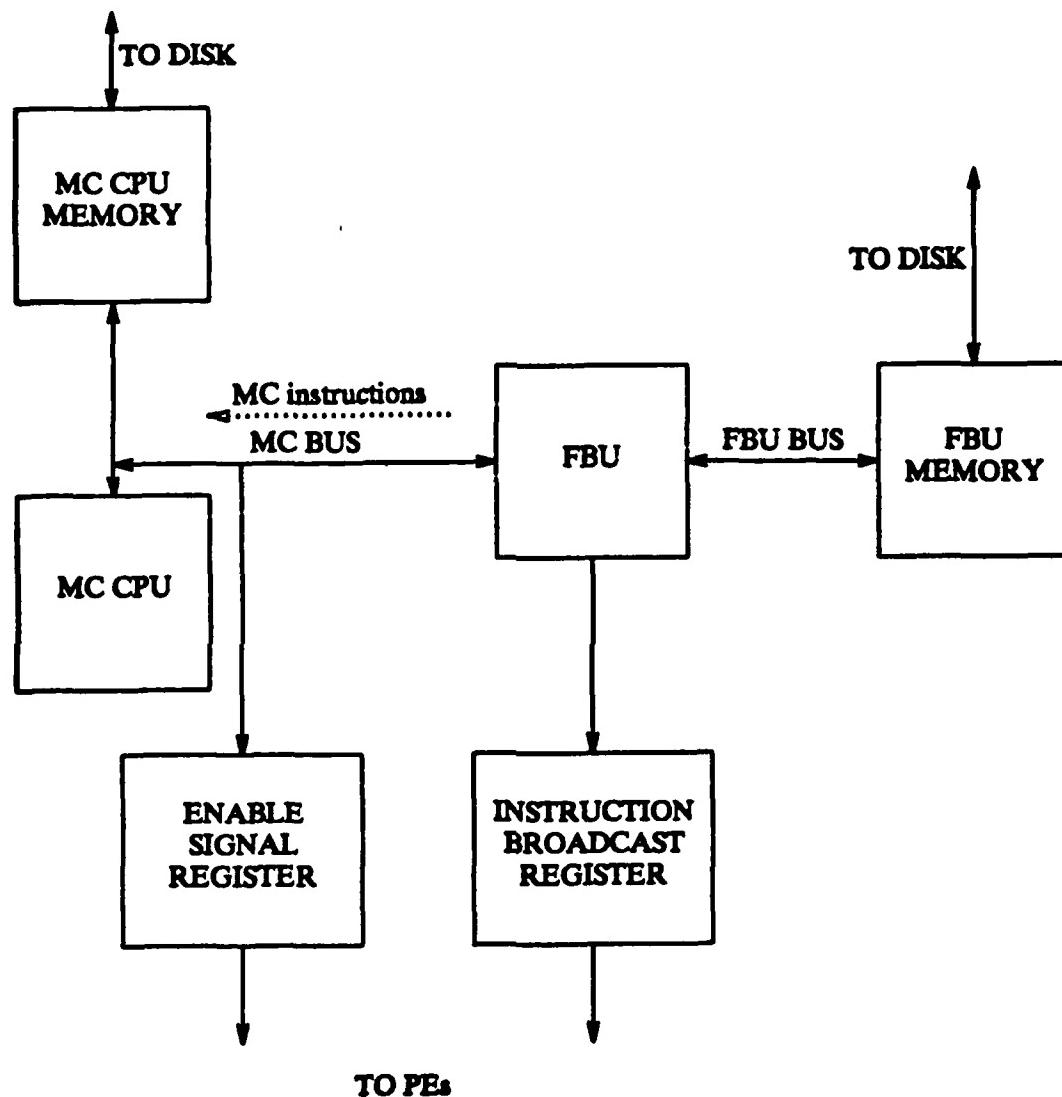


Figure 1.6.3. Master FBU - slave MC CPU organization.

MIMD mode while the FBU Memory contains only SIMD programs.

An advantage of the approach just described is that the FBU and the MC can be performing their tasks simultaneously. Another is that the MC CPU can be rather complicated since it is a commercial device. Since it is likely that the combined rate of MC CPU and PE instruction access will be comparable to the combined PE instruction and PE data access, the FBU Bus is not likely to become a bottleneck.

The principal drawback of this organization is evident during SIMD control flow changes (branches, subroutine calls and returns). A control flow change implies that the program counter of the "master" (in this case, the FPC) be updated. The update can be based on some data condition, i.e., a conditional branch, or can be some unconditional jump or branch. Either the FBU has to perform all control flow operations itself, passing the MC CPU only "scalar" processing instructions, or it can use the MC CPU to perform the computations. Neither approach is very flexible as evidenced by the following discussion.

Suppose that the FBU performs all SIMD control flow operations itself. This implies that the decoding of instructions is more complex and that many functions of the MC CPU would be duplicated in the FBU. Because the data conditions that determine the actions taken in conditional or unconditional branches reside in the MC CPU (e.g., processor status bits, address registers), each control flow operation that the FBU performs would have to be emulated by having it emit instructions requesting the MC CPU internal information followed by FBU instructions to test and update the FPC. In short, this is a tremendous burden for the FBU designer.

In the other approach, the FBU can allow the MC CPU to perform the control flow operations; however, this requires that the MC CPU program counter be synchronized to the FPC both before and after the branch instruction is executed. Before a conditional branch instruction, the FBU would send a "jump FPC" instruction to the MC CPU to align the two program counters (the FBU would fill in the current value of the FPC before it would send the instruction to the MC CPU). Then the conditional branch instruction itself would be sent to the MC CPU. Finally, the updated value of the MC CPU program counter would be read back into the FPC. This can be done by

latching the address that the MC CPU places on the MC CPU SIMD Instruction Bus into the FPC.

As before, a mechanism for distinguishing MC from PE instructions is required. For this organization, there are two kinds of MC instructions: those that cause control flow changes and those that do not. Within the class of MC control flow instructions, there are likely to be single-word instructions and multi-word instructions. Each type must cause a different type of processing sequence for aligning the program counters, broadcasting PE instructions, and so on. In a design for a FBU of this type that used a Motorola MC68000 as a PE CPU and an MC CPU, fourteen different processing sequences were identified [KuS82].

Although simpler than the all-custom MC CPU/FBU approach, this scheme still suffers from undue complexity, especially if an MC CPU with an asynchronous bus protocol is used. In particular, since the MC CPU acts as a slave in SIMD mode, an asynchronous bus' timeout mechanism must be disabled when access to an SIMD MC CPU instruction is attempted. Furthermore, if the MC CPU is waiting for an instruction, it cannot be interrupted and hence it is not free to respond to error reports by PEs or I/O completion messages from the SCU or Memory Management System. An MC CPU with a synchronous bus solves the latter problems, but complicates the FBU further in that it must now provide "no operations" to the MC CPU if there are no SIMD control instructions available. The FBU is further burdened by having to periodically send "jump to X" instructions to the MC CPUs to reset their PCs back to the beginning of the MC CPU instruction space, just as has to be done for the PEs. The conclusion is that the sharing of control and program counters between the MC CPU and the FBU has made this scheme too complicated.

Master MC CPU - Slave FBU Organization

This final organization is not the most conceptually simple, but is the method of choice from a hardware designer's point of view. Consider an organization in which the MC CPU is a conventional processor that acts as the "master" which completely controls a "slave" FBU acting as a peripheral on

the MC CPU Bus. MIMD instructions and SIMD/MIMD data are placed in the MC CPU Memory while SIMD instructions are put in the FBU Memory. This organization is shown in Figure 1.6.4.

The FBU responds to a certain range of addresses reserved for SIMD programs. When the MC CPU generates an address in the range, the FBU fetches the corresponding SIMD instruction word from FBU Memory and determines whether it is an MC CPU or a PE instruction (using one of the tag schemes discussed earlier). If it is an MC CPU instruction word, it is placed on the MC CPU Bus to be read and executed by the MC CPU normally. If it is a PE instruction word, it is written to the Instruction Broadcast Register; simultaneously, a "no-operation" instruction is fed to the MC CPU.

An advantage of this scheme is that control is centralized in the MC CPU: there is a single program counter in the MC CPU controlling execution and all program flow instructions are executed there. Another advantage is that such an FBU can be implemented as finite-state machine with very few states. The principal drawback is that the MC CPU executes a large number of "no-operation" instructions, one for each instruction word sent to the PEs. Therefore, the success of this scheme depends on the relative frequency of non-instruction accesses by the PEs as compared to the number of MC CPU instructions. For example, if the PEs fetch SIMD instructions 30% of the time, they are fetching data or MIMD instructions the other 70% of the time. Assuming the MC CPU and PEs have similar cycle times, so long as the SIMD control operations absorb no more than 70% of the total cycles the MC CPU has available, the PEs will not be starved.

Master MC CPU - Master/Slave FBU Organization

Finally, consider a variation of the scheme just presented. The organization of Figure 1.6.4 is used, but the FBU Memory contains only PE instructions for SIMD programs. MC CPU Memory contains MIMD programs, SIMD control instructions, and SIMD/MIMD data. In this variation, whenever a block of one or more PE instructions is to be executed, the MC CPU writes the block's starting address and length (number of instruction bytes) to internal registers of the FBU. The FBU carries out the fetching/broadcasting duties in

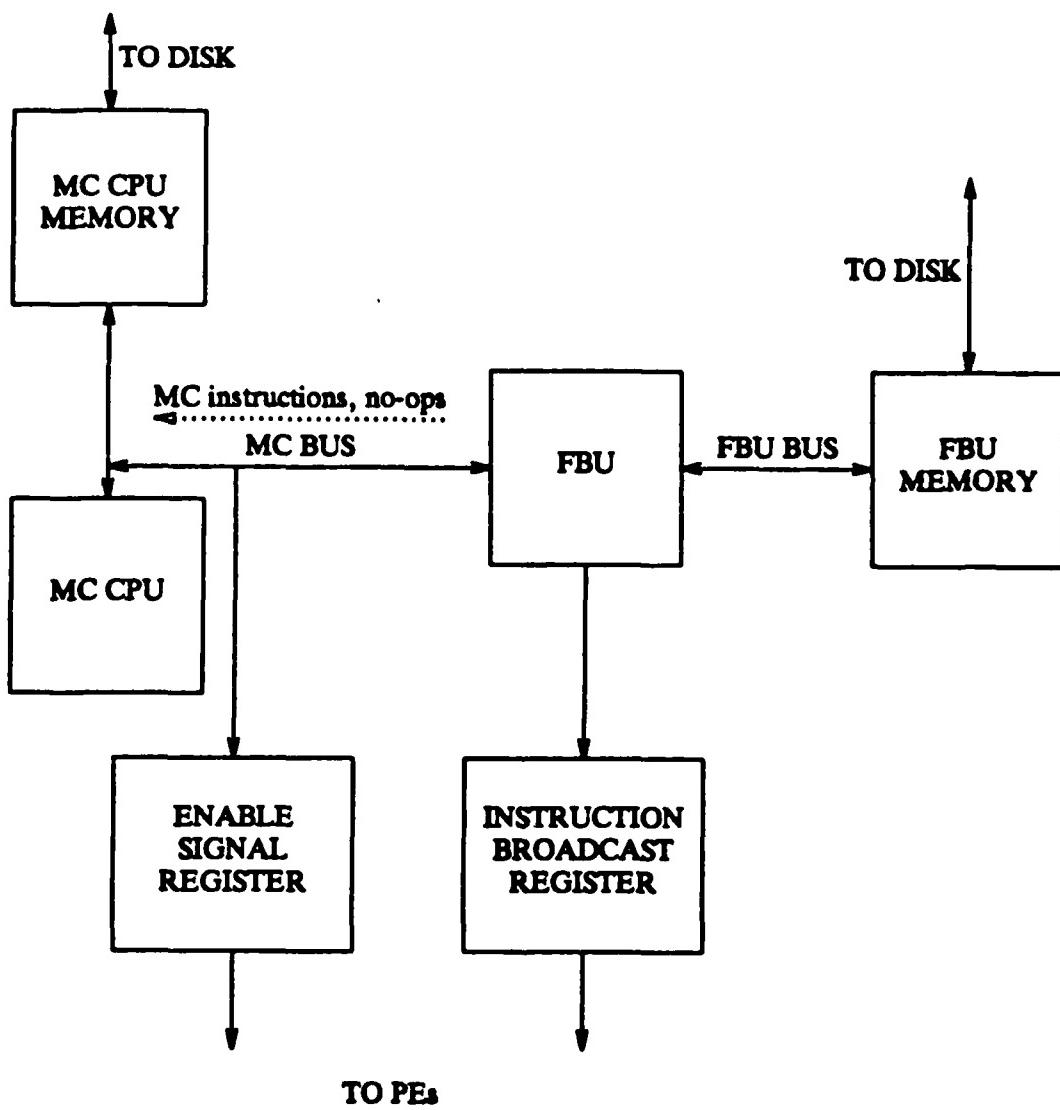


Figure 1.6.4. Master MC CPU - slave FBU organization.

parallel with subsequent MC CPU instructions, thus improving throughput. This FBU can also be implemented with a finite state machine with a relatively small number of states. It is referred to as a master/slave FBU because it acts as a slave to the MC CPU by accepting its directions to fetch blocks of instructions but acts as a master while autonomously fetching instructions from FBU Memory.

One drawback to this scheme is that the SIMD program, which consists of both MC and PE instructions, resides in two separate memories. This places a greater burden on the system assembler/loader because complete SIMD object files are no longer loaded in contiguous memory locations. The number of instructions the MC CPU executes is somewhat larger than for the "Combined MC CPU/FBU" and "Master FBU - Slave MC CPU" schemes described in the previous subsections due to the special FBU controlling instructions, but smaller than the "Master MC CPU - Slave FBU" scheme. However, no "tagging" mechanism to distinguish MC and PE instructions is needed at execution time: MC and PE instructions are implicitly distinguished by the memory in which they reside.

As shown in Figure 1.6.5, the FBU for this organization consists of a program counter (FPC) which is initialized by the MC CPU to the address of the start of a block of PE instructions in FBU memory. It also contains a count register, initialized by the MC CPU to the size of the block of PE instructions to fetch and broadcast to the Instruction Broadcast Register. The internal finite state machine sequences the machine: the FPC is placed on the FBU address bus, the fetched instruction word is latched into the Instruction Broadcast Register, FPC is incremented, and count is decremented. The cycle repeats if the count is non-zero.

Unlike the other organizations, the master/slave FBU must be polled by the MC CPU just like a peripheral device before "work" is given to it. If the FBU is busy fetching and broadcasting instructions to the PEs, it could temporarily accept no more "work," thus blocking the MC CPU's attempts to write a new starting address and count to its internal registers. A blocked MC CPU would be undesirable since it could not respond to interrupts. Eventually, the bus timeout mechanism would terminate the bus cycle and report an error, but this might not have been the correct action to take. Consider an

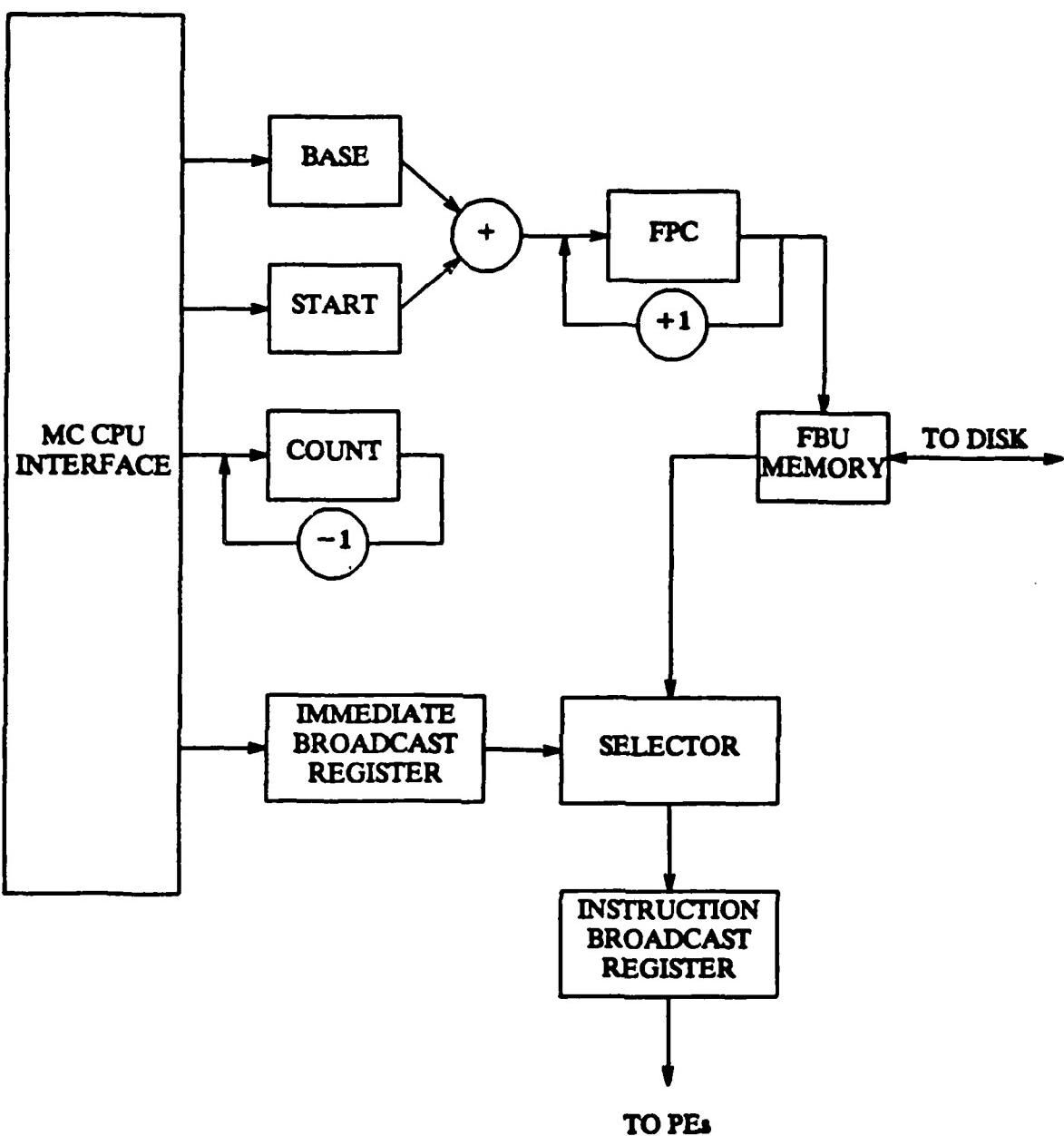


Figure 1.8.5. Internal master/slave FBU organization.

SIMD program that at some point causes the PEs to go into MIMD mode for an extended period of time. Since PEs are fetching their instructions locally rather than getting them from the FBU, the FBU never completes the broadcasting of the current "block" of PE instructions. Thus the MC CPU timeout is not an error at all but is due to "normal" program operation. The conclusion is that if the MC CPU is acting as the master, it must not be allowed to block.

Immediate Broadcast

Up until now, the actions of the FBU that have been described have been involved with fetching and broadcasting PE instructions that appear in the SIMD instruction stream. There are occasions when the MC CPU will need to broadcast some scalar quantity to the PEs or to insert a run-time dependent sequence of instructions. For this purpose, an additional feature of all of the FBUs described is an *Immediate Broadcast Register* which accepts data or instruction words written by the MC CPU and passes them unmodified to the Instruction Broadcast Register. Of course, to maintain the integrity of PE instructions, being fetched by the FBU, any multi-word or block fetch in progress is completed before the word(s) written to the Immediate Broadcast Register are passed along to the Instruction Broadcast Register.

As a simple example of the use of the Immediate Broadcast Register, consider the broadcast of a word of scalar data to the PEs at some point in the program. Since PEs expect to receive instructions rather than arbitrary data, it is the responsibility of the MC CPU to generate the object code for the instructions the PEs are to perform. In this case, an MC CPU would generate the opcode for a "load immediate word" instruction, write it to the Immediate Broadcast Register, and next write the actual word of scalar data to the Immediate Broadcast Register. In this way, the MC CPU can tailor-make instructions to suit any purpose and to modify any internal PE register or memory location.

As discussed earlier, the MC from time to time needs to reset the PE Program Counters back to the beginning of the SIMD Instruction Space. For this application, the MC CPU can use the Immediate Broadcast Register to send

the PEs a "jump" instruction. However, this implies that the MC CPU has a way to determine when the PE Program Counters need to be reset. One way would involve further decoding of the PE Program Counter values so that addresses near the end of the SIMD Instruction Space would be detected. The "OR" of the signals detecting the need for reset would be used to interrupt the MC CPU causing it to emit a jump instruction in the interrupt handler. Another scheme would be to count the number of instructions that are sent to the PEs through the Instruction Broadcast Register. When the count reached a certain value, the MC CPU could be interrupted. This has the advantage of limiting the hardware required to the MC. Still another possibility would be practical if the SIMD Instruction Space were large: a timer could be initialized to interrupt the MC CPU periodically so that the jump instruction could be sent. The later approach would probably lead to more jump instructions than necessary being issued and especially so if the PEs often went into MIMD mode for extended periods of time.

Local MC Memory

As described in the general PASM design, each MC is to have two local memory units so that loading of programs can be overlapped with execution. The comments in the section on the PE design about the need for local variable storage for applications that cannot fit in a single memory unit, the desirability of a dual arbitrated-access memory, and the use of a parallel data path between the secondary storage device (Control Storage) and MC memory apply equally to the MC memory design.

The size of MC memory is highly application-dependent. Nonetheless, some decision must be made that meets the requirements of most algorithms. Even very sophisticated SIMD programs rarely exceed a few dozen kilobytes, while MIMD operating systems programs may often achieve this size or larger. Therefore, operating systems programs will likely be the overriding memory-user in the MC. Since the size of an application or operating system program is not a function of the number of PEs used, the minimum memory requirements are fixed. The organization of the MC determines the memory requirements as well. For example, an FBU memory which stores complete SIMD

programs must be larger than one that stores only the PE instructions. If a combined memory is used, it must approximate the total sizes of the largest operating system and SIMD programs since both types of programs may be resident simultaneously.

Shared Memory

Private (local) MC memories are effective due to the large percentage of local instruction and data references as compared to global references for SIMD and MIMD control functions. On the other hand, there are applications and algorithms for which it is desirable for one MC to have access to data in other MCs' memories. For example, in a partition involving multiple MCs, the calculation of a global "if any" result is based on the combination of local "if any" results, one from each MC. Another example is an operating system program that requires shared data for synchronization or for control purposes.

In SIMD mode, inter-MC communications are inherently structured by the single instruction stream program. Consider the communication of data or control information in PASM MIMD mode. To obtain a data item from a remote MC, a transaction request must be generated, encapsulated in a message, and sent to the remote MC. When the remote MC replies, the returned message is decoded and handled. As in the PE design discussion, MCs can emulate shared memory in software by forming messages explicitly, rely on access to a special area of memory to detect a shared reference, or employ a special-purpose NIU.

The original PASM design does not call for MCs to be connected by a network as the PEs are. Instead, MCs use the SCU as an intermediate node in the communication. This decision is based on the notion that the MCs need to communicate much less often than PEs and that much of the remaining communication can be avoided by having special hardware for synchronization and comparison of 1-bit results. However, consider the advantages of a bidirectional multistage combining network connecting the MCs in PASM. For this network, the cost of the shared memory reference is lower than if the SCU is used as an intermediate link. The requested data is returned along the bidirectional link already established by the incoming request.

The use of the MC-MC combining network allows simultaneous access of a single MC memory location by several processors without serialization. A simultaneous request by many MCs for the same address is not a rare occurrence at all. Consider the use the combining network to calculate the global "if any" conditional in a partition involving M MCs. Assume that there are two shared memory locations, COND and SYNC, each initialized to zero. MCs may reach the point in the program where the condition is to be calculated almost simultaneously. Upon reaching the synchronization point, an MC calculates its local "if any" result and if true, performs a "Fetch-and-Add(COND, 1)" operation which uses a read-modify-write memory cycle to increment the shared COND variable by one. MCs for which "if any" is false need not modify COND (they could perform Fetch-and-Add(COND, 0) but this is unnecessary). Then each participating MC performs a "Fetch-and-Add(SYNC, 1)" to indicate that it has passed the point where it could have modified COND. When each of the M MCs does this, SYNC will have the value M. Each of the participating MCs repeatedly checks SYNC and when found to be M, each knows that COND reflects the global "if any" condition: zero "if none" and non-zero "if any." The calculation of other global conditions as well as simpler hardware to perform this synchronization is discussed later.

Memory Management and Virtual Memory

Memory Management concepts for PEs were discussed in the previous section. As for the PEs, the use of mapping and protection schemes can aid in higher memory utilization, program error detection, and enhanced memory fault-tolerance in the MCs. A virtual memory scheme for the PASM MCs is also desirable. In SIMD mode, MCs are close to being instruction-synchronized (they are not exactly synchronized because interrupt and exception handling can occur). If virtual memory were available in SIMD mode and one out of M MCs faulted, that MC would have to resolve the bus fault, wait for secondary storage to retrieve the missing page, and re-run the bus cycle. The other M-1 MCs participating in the partition need not wait, although it is very likely that they will fault on the same instruction or data location within short order. Therefore, it would be useful if an *anticipatory prefetch policy* [HwB84] was

used so that a page fault in one MC would cause the same page to be made available for all MCs. In MIMD mode, the situation is less critical because one MC processing a fault does not delay others directly. However, paging may overtax Control Storage due to many MCs sharing it.

As discussed earlier for the PEs and MSUs, the sharing of Control Storage among MCs implies that the overhead of each secondary storage operation should be kept as low as possible. Large page sizes increase the expected time between faults so long as there is a reasonable degree of program and data locality [Den70]. This in turn reduces the number of context switches. The implication is that the MC primary memories should be as large as is practical and affordable to decrease the number of accesses to secondary memory needed during program execution.

Instruction and Data Caches

Consider the use of instruction and data caches in PASM MCs. Unlike the situation of PEs in SIMD mode, use of an MC instruction cache in SIMD mode would significantly increase performance for most of the MC organizations discussed earlier. The reason is that SIMD programs from the point of view of the MC have a significant amount of looping - the characteristic associated with good instruction cache behavior.

In the combined MC CPU/FBU organization, the instruction cache would speed access to both the MC CPU and PE instructions in SIMD mode. It also aids performance in MIMD mode.

The "Master FBU - Slave MC CPU" organization cannot use an instruction cache for the MC CPU instructions since the MC CPU in this case operates like a PE does in SIMD mode: it never loops. However, an instruction cache may be used by the FBU for this organization to speed access to both the MC CPU and PE instructions in the FBU memory. Also, the MC CPU could use an instruction cache between it and its private MC CPU memory in MIMD mode.

The "Master MC CPU - Slave FBU" organization can make use of a cache in SIMD mode only if it is placed between the FBU and its memory. If the MC CPU had a cache for instructions received from the FBU, "no-operations" that

it received (when the FBU fetches a PE instruction) would be executed internally by the MC CPU and a new fetch request would never be seen at the FBU. This would result in PE instructions being fetched and broadcast by the FBU once, but not again if the corresponding address was in the MC CPU cache. Again, the MC CPU could use an instruction cache for MIMD instructions coming from the MC CPU memory.

Finally, the "Master MC CPU - Master/slave FBU" organization can use caches between the FBU and its memory as well as between the MC CPU and its memory. The latter cache can also be used in MIMD mode.

Data caches are desirable in the MCs for both SIMD and MIMD mode operation. Cache misses do not directly penalize other MCs in MIMD mode; therefore data caches can be used effectively so long as the cache coherence problem is addressed. The cache coherence problem causes little difficulty so long as the MCs are not connected by an interconnection network with NIU servers.

Masking Operations

One of the specialized operations an MC performs in SIMD mode is the decoding and manipulation of masks. There are several reasons why masking operations are performed in the MC rather than the PEs. First, unnecessary hardware duplication is eliminated when masking hardware is consolidated in the MC. This simplifies the design of the PEs since the hardware related to parallelism that is needed on the PE "boards" is reduced. Also, PE address masks are most easily decoded in a centralized location. Finally, masks can be manipulated in the MC independently of any operations going on in the PEs. Thus masking operations and PE operations can be performed simultaneously.

The hardware associated with masking operations may range from a set of simple external registers to a sophisticated stand-alone *Masking Operations Unit*. The minimum configuration consists of two external registers, the Enable Signal Register and the Data Conditional Mask Register. Here, the Enable Signal Register holds the current enabled/disabled status of the PEs controlled by the MC. Each time an instruction is broadcast to the PEs, bit i of the N/Q-bit register is provided to PE i to enable or disable it for that instruction. The

actual determination of the general mask bits appropriate to the MC, the decoding of PE address masks, and any AND-ing or OR-ing of masks would be performed in software in the MC CPU. Since for PASM, N/Q is at most 32 or 64, internal calculations are not especially costly. To obtain status information from the PEs, the Data Conditional Mask Register is read. Reading this port and testing its contents is not an extraordinarily expensive operation to perform in software so long as N/Q is on the same order as the internal MC CPU word size.

As discussed earlier, three different types of masks can be used by an SIMD program: general masks, PE address masks, and data conditional masks. For a partition involving N PEs, an N-bit general mask is specified; however, only N/Q of the bits are appropriate for any one participating MC. Therefore, each MC obtains the general mask bits for its PEs by indexing into them using the MC number as an index: logical MC 0 selects the first N/Q bits, logical MC 1 the next N/Q and so on. This assumes that the compiler has arranged the general mask in memory such that groups of consecutive N/Q bits correspond to the physical PEs associated with a single MC.

The need for a MC CPU to determine its physical number in SIMD mode has just been demonstrated. The need also exists in MIMD mode so that an MC can determine which PEs it is controlling. Techniques for communicating an MC number to the MC CPU either by on-board hardware or at boot-up time by the SCU were described earlier.

A PE address mask is a $2n$ -bit encoding of some of the most useful general masks. The enable/disable vectors that it can specify were described earlier. Use of such masks can significantly reduce the size of an SIMD program since only $2n$ bits rather than N bits are used to specify an enable/disable vector. However, PE address masks are inconvenient and costly to decode in software. A hardware PE Address Mask Decoder for these types of masks is desirable. Such a decoder takes as input the $2n$ -bit mask and the logical MC number and outputs the N/Q-bit enable signal vector appropriate to the MC. Since the $2n$ -bit PE address mask is at most 20 bits for 1024 PEs, it can be written by the MC CPU to an external register at the input of the mask decoder. An N/Q-bit register at the decoder output can then be explicitly read by the MC CPU.

An additional masking operations function is the maintaining of a stack of masks generated by nested "where" conditionals and PE address masks. It is the mask on the top of this stack that represents the current enable/disable signal "context" and which is to be written to the Enable Signal Register. The details of stack operations and the interplay between SIMD programs and masks are discussed in [CIS83, SiK81] and will be considered in Part II of this thesis.

Of course, all of these masking-related components could be integrated into one functional unit that would be controlled by the MC CPU. It would, of course, speed up the masking operations since it could implement a whole sequence of functions formerly requiring several MC CPU instructions. However, it duplicates some of the functions already available in the MC CPU such as the ability to store, stack, and perform logical operations on masks. In addition, its stack would be fixed-size, limiting the nesting level of conditionals in SIMD mode. It is for this reason that the "full-function" masking operations unit complicates rather than simplifies the design of an MC, both from a hardware and operating-systems-programming point of view.

Inter-MC Communication and Synchronization

In multiple-MC configurations, the PEs of two or more MCs may be combined to form larger machine partitions. Synchronization of the MCs is necessary in SIMD mode before certain operations such as interconnection network transfers and the determination of the "if any" condition. In the case of network transfers, a time skew between PEs controlled by different MCs might cause conflicts to occur in the network switching nodes. For "if any" testing, the cooperating MCs must first determine their local "if any" condition, synchronize, compare results with the other MCs to determine the global "if any" condition, and then act on that condition before continuing.

Because of the way multiple PASM MCs are combined into partitions, the global "if any"-type conditions can be calculated by a "tree" of simple hardware. Figure 1.6.6 shows that each MC needs an I/O port for which there is an output "local result" bit and $q+1$ "global result bits," one for each of the $q+1$ different-sized machine partitions the MC can participate in. Global

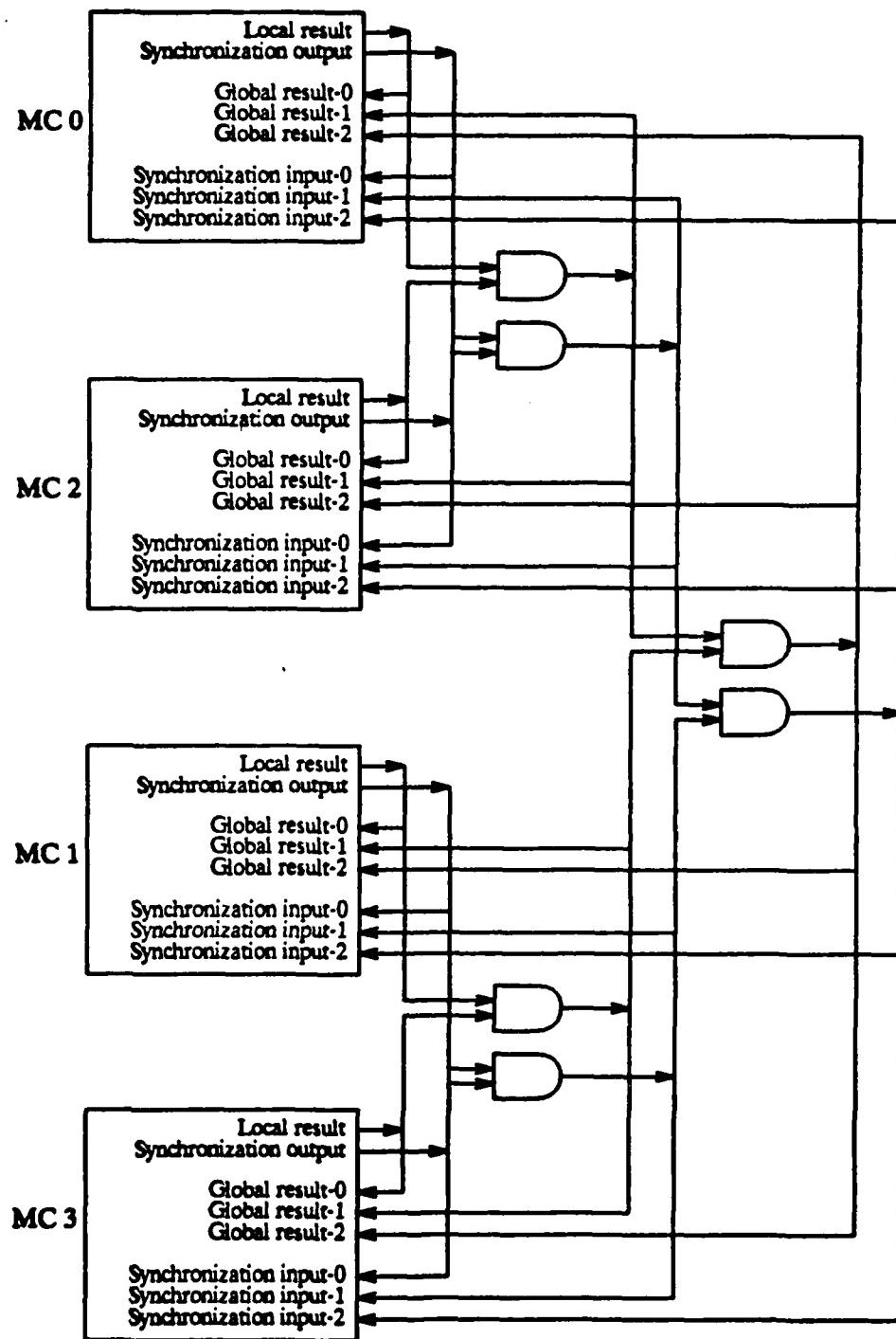


Figure 1.6.6. MC global status communication tree.

result bit r , $0 \leq r < q+1$, corresponds to a machine partition size of 2^r MCs. Technically, global result bit 0 (machine partition size of one MC) is unnecessary; however, it is included to make the synchronization procedure generic.

To calculate the global "if all" condition, the local "if all" condition is written to the "local result" bit. A "1" is written if "if all" is true; otherwise a "0" is written. If the appropriate "global result" bit is a "1," the global "if all" condition is true. When testing the "if none" condition, a "1" is written if "if none" is true; otherwise a "0" is written. If the result is "1," the global "if none" is true. The result of "if any" is the complement of the "if none" result.

Because MCs operate independently, the global results calculated using the scheme described above cannot be depended upon unless the MCs are synchronized before they read the global results. The synchronization can be implemented using a software rendezvous approach. As each participating MC reaches the synchronization point, it asserts its "synchronization output bit." The CPU then polls one of the $q+1$ "synchronization input bits," whichever is appropriate for the machine partition size the MC is participating in. When the synchronization input bit is found to be asserted, indicating that all MCs have reached the synchronization point, the MCs are instruction-synchronized and they negate their "synchronization output bit" to prepare themselves for later synchronizing steps. The MCs can then broadcast PE instructions to use the network, read "global results," etc. An alternative to providing all $q+1$ bits to MCs would be to have the SCU control the tree logic combination (based on its knowledge of the partition combination) and to return only one condition bit and one synchronization output bit to each MC.

Another method for combining results among multiple MCs of a machine partition was described in [SiM78b]. Here, a 1-bit data bus constructed with "wired-and" (open-collector) logic was used in conjunction with a $(q+1)$ -bit "job ID" bus that arbitrated usage of the data bus. Since it was determined that at most $2Q$ SIMD jobs could be active simultaneously (one in each memory unit of the MC), $q+1$ bits could encode a "job ID." When an MC required global information, it requested bus mastership of the data and job ID buses. It then would broadcast its job ID which the other MCs would compare to their own job IDs. All MCs that matched the ID placed their local results on the 1-bit data bus simultaneously so that each could read the global result.

There are two problems with this synchronization scheme. The first is that a global bus connecting the MCs is used. This allows MCs of a job that are badly out-of-sync to "hog" the bus until they synchronize, preventing other jobs from sharing results. This violates one of the basic tenets of PASM: that the system be partitionable so that two unrelated jobs do not conflict with each other. The second problem is that the assumption about the range of job IDs is based wholly on the "double-buffered" memory concept. For a research machine, in which new control and operating systems ideas are to be tried, the limited job-ID bus width proves too restrictive. Conceptually, there should be no limit to the number of jobs that each MC could have active.

Another architecture concept that seems to have limited utility for PASM is the "reconfigurable shortable MC-MC memory bus" first proposed in [SiS81b]. It allows the sharing of memory modules by the MCs in a machine partition. The MCs are connected to the memory modules using the scheme shown in Figure 1.6.7. The MCs are ordered on the bus in terms of the bit reverse of their addresses due to the PASM partitioning rules. The desirable attributes of this architecture as stated in [SiS81b] are that more program space for jobs using multiple MCs would be provided and that it provides a degree of fault tolerance since known-faulty memory modules could be ignored.

This reconfigurable bus structure can be attacked on several grounds. Being a global structure, its scalability is questionable. Suppose that all MCs are participating in a single machine partition. If the "next" instruction is required from MC memory module 0 (Figure 1.6.7), it must encounter $Q-1$ propagation delays due to the "through connection" switches before arriving at MC 7. If the propagation time per shortable switch is 10 ns, for a larger PASM system with $Q=32$, 310 ns are required. This exceeds the access time of even fairly slow dynamic memories. Furthermore, this assumes that the address provided to the MC memory module was produced "locally." If the "logical MC 0" is always responsible for generating addresses, the instruction access time is correspondingly increased for references to distant memories.

If the MCs and MC memory modules were only pairwise-connected, still allowing a degree of fault tolerance, the propagation times would be much more reasonable. However, it is useful to re-examine the reasons why such a reconfigurable structure was advocated in the first place. Clearly, the fault-

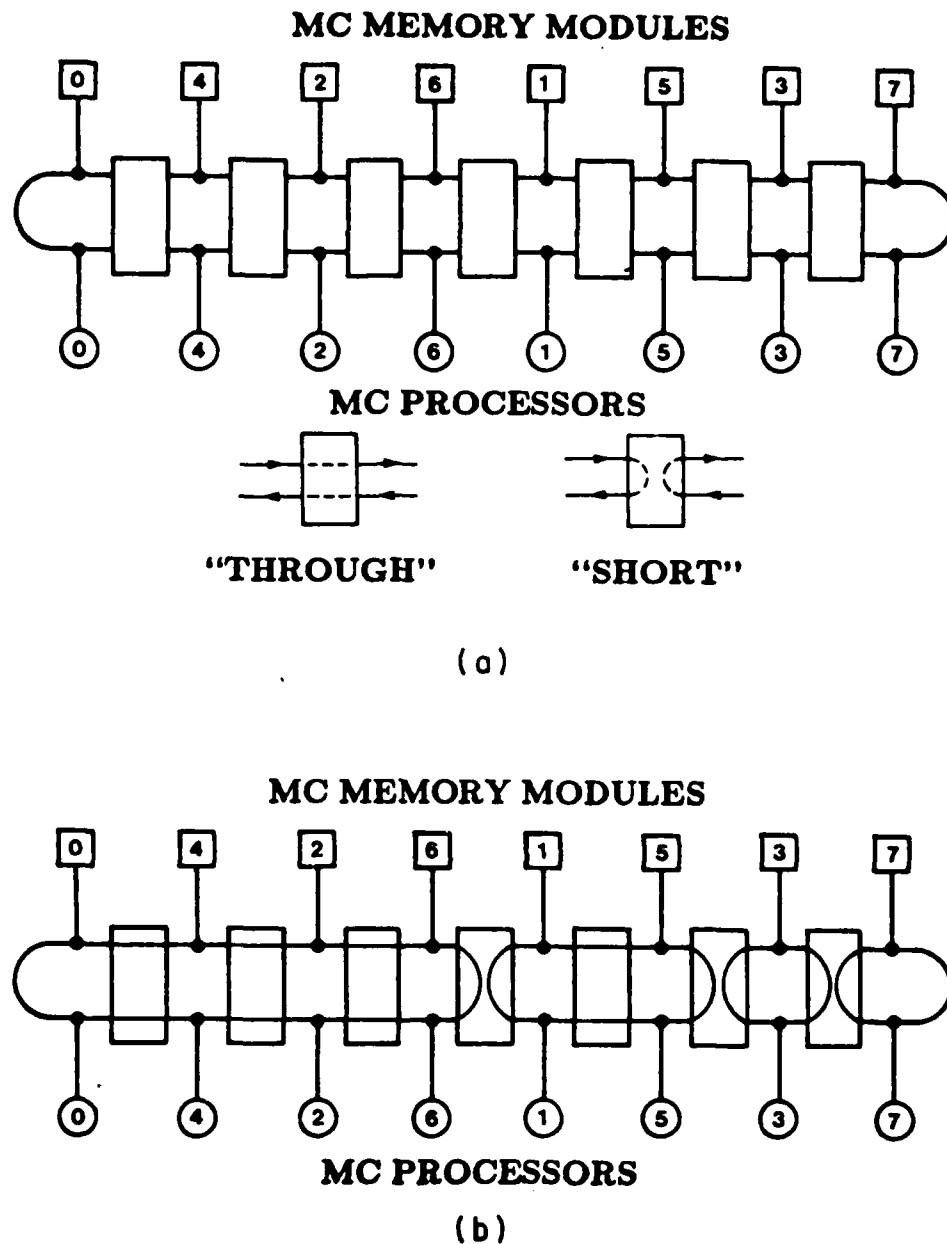


Figure 1.6.7. (a) Reconfigurable shared bus scheme for interconnection MC processors and memory modules, shown for $Q=8$, where each box can be set to "through" or "short." (b) Bus set for MC 0, 2, 4, and 6 forming one machine partition, MCs 1 and 5 forming a second machine partition, MC 3 forming a third machine partition, and MC 7 forming a fourth machine partition.

tolerance aspects of the scheme are appealing. But the CPU-memory interconnection is very reliable in practice: if it is not an on-board connection, it uses a fixed backplane which is almost as reliable. It is cabling and off-board connectors that are the most failure-prone; thus, the use of this scheme to connect MCs with physically distant memories would probably decrease reliability, not increase it.

The one aspect of the reconfigurable bus that does have some merit is that it provides more memory space. However, memory prices are becoming so ridiculously low that this should not be an overriding concern. Given that the design and implementation cost involved with the reconfigurable shorting bus probably exceeds the cost of doubling or even quadrupling the MC memory capacity, the memory size issue is no longer so important. From a philosophical standpoint, it does not seem logical to assume that a job that could run on one MC would require more memory if it ran on several. Stated in another way, if a job required more memory than was available on a single MC, it should not be necessary to use more MCs. Rather, the memory requirement would be handled using overlays or virtual memory concepts so that the compilation, assembly, and loading of a program is independent of the desired machine size.

Other MC Specialized Components

The remaining parts of the MC are rather conventional: for communicating with the PEs, SCU, and Memory Management System, serial or parallel I/O port connections are required. As discussed previously, the MC-PE connections are used in SIMD mode by the PEs to inform the MC of any internal fault or error. The MCs use these connections in MIMD mode to schedule processes onto processors and for other control functions. In either mode, the MC can send a signal to the PEs to cause them to reset to a known state. This is important at boot-up time or after a PE error that the PE cannot recover from itself.

The MC-SCU connection is used in SIMD mode to handle job scheduling requests and completion acknowledgments. It can also be used to inform the SCU of any internal MC fault or error. In MIMD mode where a machine

partition of more than one MC is used, the SCU acts as a clearinghouse for inter-MC scheduling information. In either mode, the SCU can send a signal to the MCs to cause them to reset to a known state. This is important at boot-up time or after an MC error that the MC cannot recover from itself.

The MC-Memory Management System connection is used during the execution of SIMD or MIMD jobs to request the loading/unloading of files from/to secondary storage. More details on this interaction are given in a later chapter dedicated to memory management.

For the expected low communication traffic on the MC-PE, MC-SCU, and MC-Memory Management System links (as compared to the inter-PE traffic, for example), a number of implementation options are available. Serial data channels are less expensive than parallel ones in terms of board and connector space but are inherently slower. Also, serial channels are typically more resistant to noise for equivalent conductor shielding and can span greater distances than can parallel channels.

Another consideration is that reliable communication on the channel must be ensured: inter-processor messages must not be lost or corrupted. Recently, a number of hardware device controllers have come to market that ensure reliable delivery of messages. Such devices typically accept a destination address and a byte stream from a processor and break the stream up (if necessary) into a sequence of packets. To each packet is added an error detecting/correcting code, a sequence number to detect lost packets, and other information identifying the sender and receiver. At the destination, packets are decoded, checked for errors, and sequenced in the order they were sent. If packets fail to be acknowledged by the destination after a reasonable time, the device controller at the source automatically attempts retransmission. The use of such devices to control communication over serial and parallel links that may be prone to error reduces the burden on the CPU to perform the packet formation, error checking, and retransmission. Such device controllers are highly recommended for a system such as PASM.

1.6.5 Overlapping Schemes

Since the MC, PEs, and interconnection network are all capable of independent operation, the operations of these components can be overlapped resulting in increased machine performance. Overlap allows the MC, PEs, and network to perform their own tasks, synchronizing only when there is information to be exchanged. Overlapping can be improved as additional hardware (e.g., latches, queues) is added at the interfaces of these components. This section presents details of several different MC - PE and MC CPU - FBU overlap strategies. Although they are given as "cases" of increasing overlap, increased overlap does not always enhance performance due to the cost of enqueueing/dequeueing operations and flushing of the "pipeline."

MC - PE Interconnection

MC - PE overlap was originally discussed by the author in [Kue81, KuS81]. It is briefly described here to complete the survey of implementation alternatives for PASM.

The MC organizations presented earlier each assumed the use of a single Instruction Broadcast Register. This allows the MC to fetch instructions or to execute MC CPU instructions while the PEs are executing. However, the MC must wait until the PEs have completed their operation before broadcasting the next PE instruction.

Additional overlap can be obtained by using a FIFO instruction queue in the MC whose output feeds the Instruction Broadcast Register. This allows the MC to send PE instructions (opcodes and operands) to the queue without having to wait for the PEs to complete their current instruction. A copy of the N/Q-bit Enable Signal Register is stored in tandem with the opcode/operand when the queue is written. This set of enable signals must be copied to the queue since MC masking operations (changing the Enable Signal Register) might be performed before the queued PE instruction is actually executed. The Illiac IV and MPP control units and PEPE arithmetic control unit use instruction queues; however, enable signals are not queued with each instruction. Rather, the queue is emptied before a control unit masking operation is performed.

Simulation studies of MC - PE overlap were performed using several test algorithms [KuS81, SiK81, SiK82, KuS82]. As might be expected, the queue improved the performance of the system because it allowed the MC to fill the queue, keeping the PEs "satisfied" even when there were several MC instructions in a row to be executed. However, the queue rarely led to the MC and PE operations being completely overlapped. Frequently, programs began with a large number of consecutive MC initialization steps during which time the PEs are idled. Nonetheless, 30-40 percent improvements in execution time were obtained through the use of a queue capable of holding 32 16-bit words [KuS82]. The "best" length of the queue is program-dependent: if the queue is of sufficient length to keep the PEs from starving for instructions during the longest string of consecutive MC instructions, making it longer will not improve performance.

The PEs within a partition synchronize in SIMD mode using the following protocol. Each time a PE is ready to accept the next SIMD instruction, an "instruction request signal" is sent to the controlling MC. The N/Q signals are AND-ed (using an open-collector circuit) to form an MC-group instruction request. To keep the multiple MC-groups synchronized, the MC-group instruction requests are further AND-ed to form a common instruction request. When this common request is true, meaning that all PEs in the partition are waiting for the next instruction, an instruction (if available) is broadcast by the MC(s). The instruction is accompanied by an acknowledge signal indicating the presence of the instruction on the bus. The PE hardware that intercepts SIMD instructions ensures that disabled PEs are always "requesting" instructions.

Since it is possible that some MCs in a partition will not have an instruction available to be broadcast when one is requested, while other MCs will, acknowledge signals from the MCs are also combined to form a "common" acknowledge. This is so that all of the PEs in a partition see the acknowledge signal at the same time to keep them cycle-synchronized as well as instruction-synchronized.

MC CPU - FBU Interconnection

The MC CPU and FBU operations may also overlapped for some configurations. For the "Master FBU - Slave MC CPU" configuration, the least overlap is obtained by forcing lock-step operation: either the FBU or the MC CPU executes, but not both simultaneously. A more practical situation allows the FBU to begin the fetching, decoding, and broadcasting of PE instructions while the MC CPU is operating. The most overlap is obtained by adding a FIFO queue to hold pending MC CPU instructions and to allow the FBU to begin a new fetching cycle as soon as it has completed the current one.

Here, the addition of overlap due to the queue yields poor results [KuS82]. The reason is that branch instructions stop the fetching process because they affect the program counter (which the FBU maintains to know the "next" instruction). The FBU must wait until the MC CPU has emptied its instruction queue and adjusted its internal program counter based on the result of the branch before continuing. Because the MC CPU stream is rich in branch instructions, little pipelining is available. The queue becomes a hindrance simply due to the time required to queue and dequeue the instructions.

Instruction "pre-fetch" buffers would be useful for improving the performance of the queue. One buffer pair would contain the next MC and PE instruction to be executed if the branch was taken. The other pair would contain the next instruction to be executed if the branch was not taken. This technique has been used extensively in main-frame architectures (e.g., CDC-6600 [Gri74]). Due to the complexity of setting up additional buffers and interfacing them with the prefetch buffer that may already be present internal to the MC CPU, it is felt that this approach would not be cost-effective.

Now consider overlap schemes for the "Master MC CPU - Master/slave FBU" organization. The least overlap is obtained by disallowing the MC CPU from performing any instructions while the FBU is fetching and broadcasting a block of PE instructions. A more reasonable approach (discussed earlier) allows the MC CPU to perform its own instructions while the FBU is operating, but prohibits the sending of another instruction to the FBU until it has completed the current block fetch. Finally, consider using a FIFO queue of pending block fetch requests for the FBU; the MC may add "work" to the queue so long as it is nonfull.

It can be argued that as long as the FBU can fetch and broadcast at a rate fast enough to satisfy the average PE instruction rate, either the FBU work queue or the MC-PE instruction queue alone would be sufficient to keep the PEs from starving during long MC CPU instruction sequences. Since each item of FBU "work" may result in the fetching and broadcasting of multiple words, an FBU work queue can be comparatively shorter than an MC-PE instruction queue to achieve the same protection against starving the PEs. Use of both queues is probably superfluous; the result would be a nearly continuous full condition of the MC-PE instruction queue.

If there is no MC-PE instruction queue (just a single Instruction Broadcast Register), the FBU needs access to the FBU Memory immediately upon demand to keep the PEs satisfied. If the secondary storage device (Control Storage) is currently servicing the FBU Memory, access may be delayed one or more cycles. Therefore, if FBU Memory occasionally suffers from access conflicts, the edge is given to MC-PE instruction queues and no FBU work queue. Otherwise, the FBU work queue is preferred because it can be shorter, narrower (it need not contain enable signals), and be accessible to the MC CPU bus. This last attribute is important during context switches: FBU work to be performed is part of the MC "state" and must be saved. The MC-PE instruction queue is comparatively inaccessible from the MC CPU; therefore, the FBU would need to save its own state during a context switch.

1.6.6 PASM Secondary Memory Systems

In a typical stand-alone computer system with a disk-based secondary storage system, the physical disk is formated, read, and written via a specialized interface known as a *disk controller*. The disk controller maps the logical blocks of data that the system CPU deals with into physical blocks that are arranged on the physical disk media. Thus the system CPU need not be burdened with the details of the particular magnetic encoding formats used, rotational characteristics, control of the track-to-track head movements, power-up and power-down sequences (to protect the media), and other considerations. The system CPU considers a disk to consist of a large number of equal-sized blocks, each of which may be read or written independently. A block is the

smallest size object that can be read or written and is typically 2^6 to 2^{10} bytes in length.

The *file system* is a logical organization of blocks imposed by the operating system. Disk blocks either contain data that is part of a file or directory information. Directories store the names of files and lists of the disk blocks that make up the files. Consecutive (contiguous) disk blocks do not necessarily contain related data: a linked set of data structures keeps track of the location of file, directory, and free disk blocks. The operating system running on the system CPU reads, writes, and manipulates file and directory blocks resulting from requests made by user programs.

In PASM, Control Storage is shared by Q MCs and the System Control Unit while each of the MSUs is shared by N/Q PEs. If the SCU, MCs, and PEs were to coordinate their access to the disk controller associated with each physical disk and to each maintain and manipulate the file system in a coherent manner, an extremely complicated set of operating system disk access rules would be needed. For this reason, Control Storage and each MSU has associated with it a CPU dedicated to file system operations. This CPU will be known as the *File System Server*. This unburdens the SCU, PEs, and MCs from having to manipulate files at the logical block level; rather, these processors refer to files by only their names and other high-level attributes such as their lengths and access permissions.

As discussed earlier, each File System Server needs to have a parallel I/O channel with DMA capability connecting it to each of the primary memories it serves. In the case of MSU File System Servers, N/Q channels are needed, one to each PE memory. Control Storage requires Q+1 channels, one for each MC and the SCU. If MC CPU and FBU Memory are separate devices, more than this number of channels would be desirable (unless both memories appear appear in the same address space from the disk side).

Requests for service arrive at File System Servers via parallel or serial I/O ports: each MC and the SCU have a port to Control Storage; one of the processors of the Memory Management System has a port to each of the MSUs. The choice between serial and parallel I/O is subject to the same considerations discussed earlier.

1.6.7 PASM System Control and Memory Management

As described earlier, the SCU and Memory Management System processors can all be conventional stand-alone computer systems that operate in a distributed fashion. Therefore the only criterion is that their processing power match the number and types functions that they will execute. Each needs a number of I/O ports so that data and control information such as process tables, file directory information, and so on can be shared.

Because PASM is a research machine, the development of its distributed operating system will undergo many transformations. Therefore, the connections between the SCU and Memory Management System processors should be developed such that they are easy to change. It is proposed that serial or parallel I/O ports connected with flexible cable be used rather than backplane connections to provide this flexibility. The bandwidth provided by parallel I/O port connections is probably greater than required, but the number of interconnections is small and will accommodate any high traffic intensities that may result from less than optimal initial operating system designs. Figure 1.4.2 indicates these I/O connections for the PASM prototype; connections for a larger PASM system would be similar.

Studies of the operations the Memory Management System processors perform and the data they share indicates that a bus connecting these processors (and perhaps the SCU as well) would be a desirable architecture. The bus would provide access to a shared memory area where global operating system tables would be stored, thus eliminating the problem of keeping multiple copies of this information. To reduce bus contention, these processors would access local memories to obtain their programs and local data and would also use the I/O ports between them for private communication. For example, file loading/unloading commands would be passed among the processors on the I/O ports while the tables indicating the commands' progress would be accessed in the global memory. Additional considerations are discussed in Part II of this thesis.

CHAPTER 7

PASM PROTOTYPE DESIGN

1.7.1 Role of the Author

Design and construction of a PASM prototype began in earnest in mid-1984 and is continuing as this thesis is being written. The author's involvement in the general design considerations for PASM is evidenced by the descriptions of system components given in the previous chapter. While the author has actively participated in the selection of an appropriate subset of "desirable" features to be implemented in the prototype, the actual implementation decisions such as the physical layout of boards, cabinets, and cabling, the choice of particular bus standards, and the detailed chip- and board-level implementation has been performed by others, notably Thomas Schwederski.

Throughout the prototype development, the author has joined internal design review teams to ensure that the hardware implementation decisions made were: *consistent with the original PASM design parameters* -- by ensuring that all of the important PASM features such as SIMD mode and the use of multiple secondary storage devices would be supported; *logically capable of functioning* -- by detecting and correcting design flaws that would prevent the hardware from operating under some set of conditions; *capable of being programmed* -- by ensuring that the prototype could use available languages and operating systems techniques and be controlled efficiently; and *in keeping with budgetary and time constraints* -- to allow the design to be executed primarily by university graduate and undergraduate students on a part-time basis, on a limited development budget, and with minimal administrative support.

1.7.2 General Design Constraints and Guidelines

The purpose of the prototype is to develop a small-scale model of PASM so that design experience can be gained, new ideas can be tested, software can be developed, and design flaws exposed before a major commitment of resources for a larger PASM system is undertaken. A major goal of any parallel computer system is the proving of its *scalability*. A machine is said to be scalable if conceiving, building, or programming it increases in difficulty or cost as a linear function of the size of the machine and that the performance of the machine increases linearly as well.

In practice, perfect *architectural scalability* is unattainable. While computing power and cost increase linearly as processors are added, when the processors are to be interconnected the designer has to make tradeoffs between degradation of performance during communication or a larger than a factor of N increase in the network cost. The choice of a fully-connected network does not degrade the performance of the system at all as N increases; however its cost increases as N^2 . A multistage Cube-type network degrades the communication performance by just $\log N$; however its cost increases as $N \log N$. Finally, a bus degrades the communication performance by N due to its limited bandwidth; yet, it can be implemented with the linearly scalable increase in cost. Since practical parallel algorithms generally spend much more time computing than communicating, designers of large systems of processors have generally used $N \log N$ or linear-cost growth networks rather than fully-connected ones because the overall performance is not degraded by the full $\log N$ or N factor.

PASM does not achieve architectural scalability in computing elements either: the hierarchical PASM organization involves $O(N \log_2 N)$ processors for a system with N PEs. Yet such an organization can be seen to have a positive effect on the implementation scalability (to be described).

Algorithm scalability is implied if increasing the number of processors used to perform an algorithm results in a corresponding increase in performance (using execution time, throughput, or some other measure). Increasing N beyond the problem size P (expressed as the number of fundamental operations) clearly does not result in a performance increase; therefore, algorithm scalability is defined so long as $N \leq P$. Many earlier studies have focused on the

use of the PASM system: algorithm complexity and algorithm simulation studies were performed to demonstrate that the parallel image and speech processing algorithms could effectively use hundreds or thousands of processors.

Implementation scalability in the narrow sense means that components can be added to a system without making changes to the existing implementation scheme. This is rather difficult to achieve in a design because cost and expediency often make a small system radically different from a larger one. An example of narrow sense scalability is a distributed computer system in which processors communicate only with a few of their immediate neighbors: processors can be added indefinitely to such a system. Implementation scalability in the wide sense means that the nature and function of the basic components does not change as the machine size changes, but the technology employed may change. For example, PEs in a small system may each be implemented with several physical boards and be interconnected using point-to-point wiring. In order to satisfy physical space limitations, PEs of a larger system would need a different implementation (e.g., a single-board PE) and a more regular interconnection scheme.

Now in more concrete terms, the specific design requirements of the PASM prototype are listed as they were communicated to the author at various times during the development period by Howard Jay Siegel. Prior to the funding of the prototype, the design requirements communicated to the author were used as a baseline design for which funding was sought.

- | | |
|-------------|---|
| May 1981 | The prototype shall consist of $N=16$ PEs, $Q=4$ MCs, $N/Q=4$ MSUs, Control Storage, SCU, and a Memory Management System. (The design is to operate in a manner described in [SiS81b] and in Part I, Chapter 4 of this thesis.) |
| June 1981 | A microprocessor such as the Motorola MC68000 should be investigated for use in prototype PEs. |
| August 1981 | A "system monitor" device shall be included for the purpose of gathering performance data. |
| May 1984 | The prototype shall be constructed at a cost not exceeding \$150,000 during the two-year period commencing May 1984. |
| July 1984 | PEs are to be connected by an Extra Stage Cube network. |

- October 1984 Every effort should be made to develop hardware that would be implementation scalable in the narrow sense to a system size of $N=64$ and $Q=4$ or 8.
- October 1984 MCs and MC memories are to be connected pairwise for fault-tolerance; the fully reconfigurable/shortable MC-MC memory bus need not be implemented.

The preceding list gives the only design guidelines that were communicated; the specific design and implementation decisions based on these guidelines were carried out completely by the graduate/undergraduate student design team. Major aspects of the design were presented to Howard Jay Siegel from time to time during the development period and were approved by him.

1.7.3 Prototype Implementation Summary

Early Design Decisions

As a result of experiences with the Motorola MC68000 CPU [Mot84a] in earlier design and simulation experiments [KuS82], that microprocessor was chosen as the basic computing element for PASM prototype PEs, MCs, and other system components. In retrospect, this has proved to be a good decision: the MC68000 has been very well received by the computer industry as evidenced by the large number of commercial products that use this processor. Compilers and support software are readily available. Also, in the meantime, Motorola has expanded the 68000 family to include more advanced CPUs and a wide variety of peripherals.

Because of the system budget and development time constraints, no custom LSI or microprogrammable bit-slice designs could be incorporated in the prototype. Therefore, the prototype was designed to use commercial boards wherever possible (within cost constraints). Special function boards, where needed, were studied carefully so that their design could be made general-purpose enough to be used in more than one place in the system. For example, if all of the specialized functions of the PEs and MCs are consolidated on one board type, that board can be used in both places. Although every effort was

made to find implementation approaches that would yield high performance, that performance could be (and often was) traded away for modularity, ease of design, testability, reduction in complexity, affordability, or any reason that would allow the prototype to be built in the stated 2-year time frame (May 1984 - May 1986).

All of the physical boards in the prototype are in the standard double Eurocard format; they are connected through the VME bus [MoM81] which is particularly suited for the Motorola 68000 microprocessor family. A standard bus has the advantage of readily-available backplanes, card cages, and accessories. Figures 1.7.1 through 1.7.4 show the complete set of physical boards developed or obtained commercially that are being used to construct the prototype. As shown, all boards requiring access to the VME bus have an "upper" 96-pin connector on the backplane side of the board. A VME bus backplane spans the boards that are connected to it. The "lower" 64- or 96-pin connector is reserved for user-defined applications. Front-panel male connections accept flat "ribbon" cable of varying widths terminated with an appropriate connector. Detailed descriptions of the pin assignments, cabling diagrams, power routing, etc. are beyond the scope of this thesis.

PE Design

The prototype PE consists of five boards: a CPU Board, two Memory Boards, an MC-PE I/O Board, and a PE-Network Interface Board. Of these, only the CPU Board is a commercial product.

The CPU Board (Figure 1.7.1) is the Motorola MVME-110 model consisting of an MC68000 microprocessor, sockets for local static RAM and EPROM, a serial port, programmable timer, bus watchdog timer, I/O channel interface, and a VME bus interface. An MC68010 CPU can be substituted for the MC68000 yielding somewhat better performance for integer multiplication and division operations and also providing support for virtual memory (re-running of bus cycles). There are no provisions for a cache on the board. Very limited memory management support exists. The local RAM memory can be user-write-protected but not user-read-protected on 2K-byte segment boundaries. The programmable timer can be user-read- and user-write-protected, but the

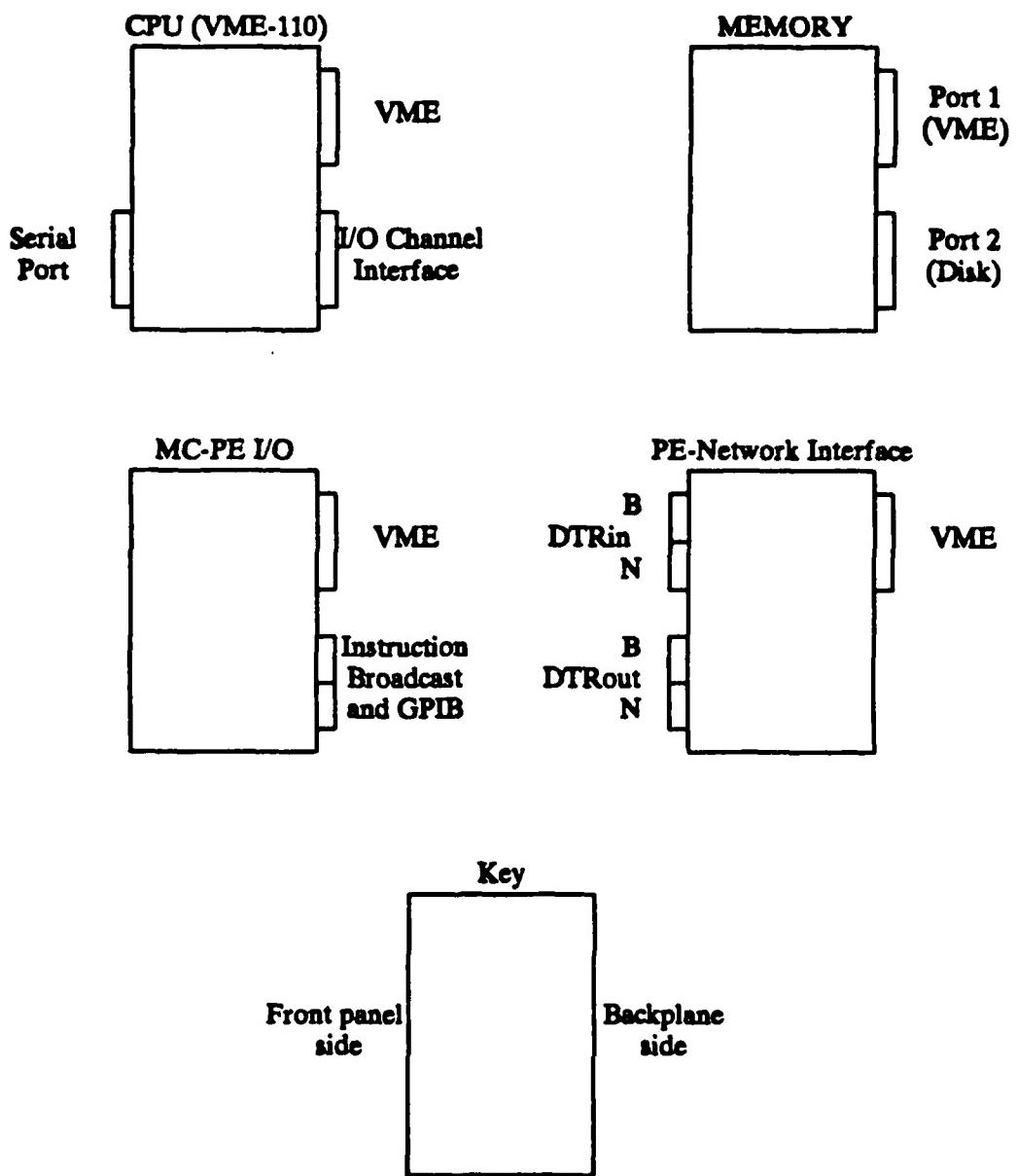


Figure 1.7.1 CPU, Memory, PE-Network Interface, and MC-PE I/O Boards. DTRin and DTRout are connected to the network input and output, respectively. B and N refer to the “bypass” and “normal” network DTR inputs and outputs.

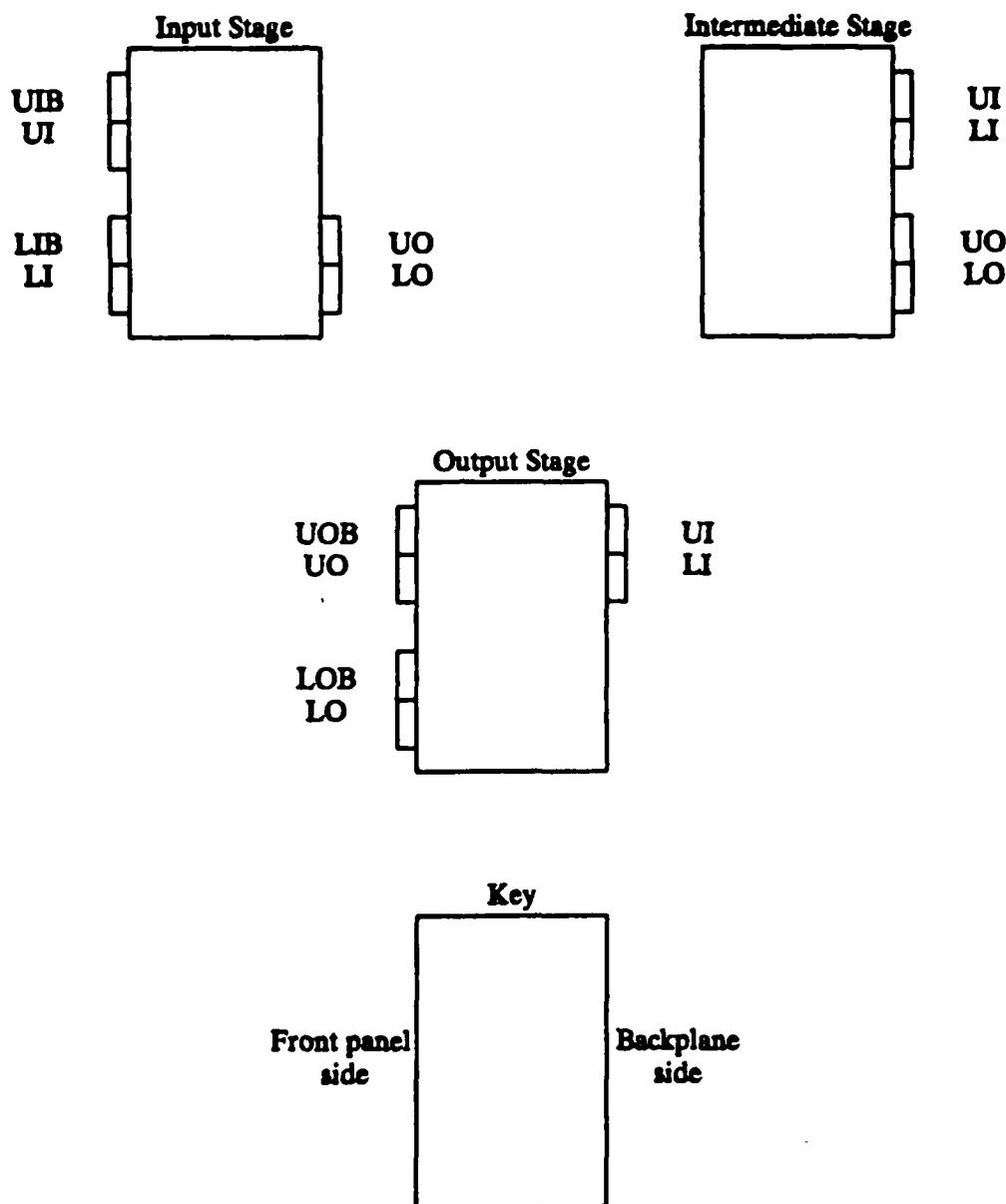


Figure 1.7.2 Network Input, Intermediate, and Output Stage Boards. UI and UO are the "normal" upper input and output, respectively. Similarly, LI and LO are the lower input and output. UIB, UOB, LIB, and LOB are the corresponding "bypass" connections.

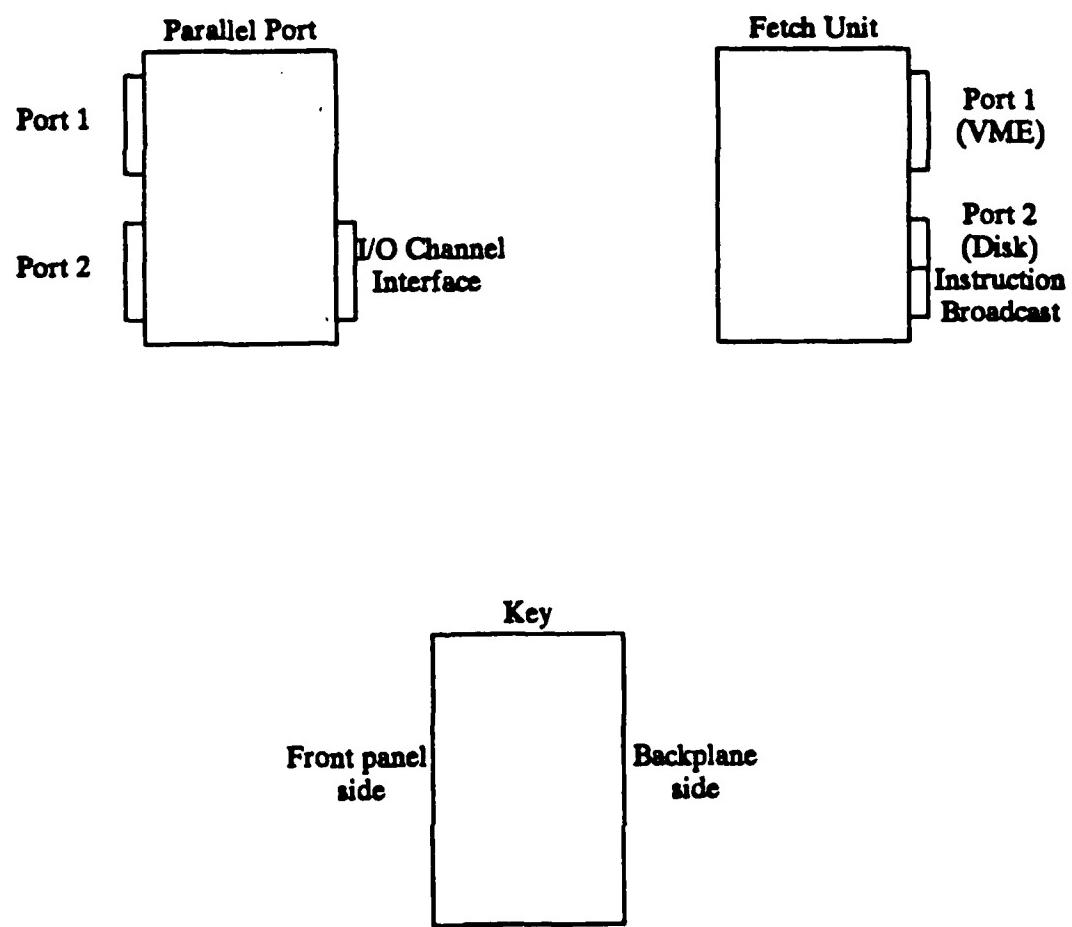


Figure 1.7.3 Parallel Port and Fetch Unit Boards.

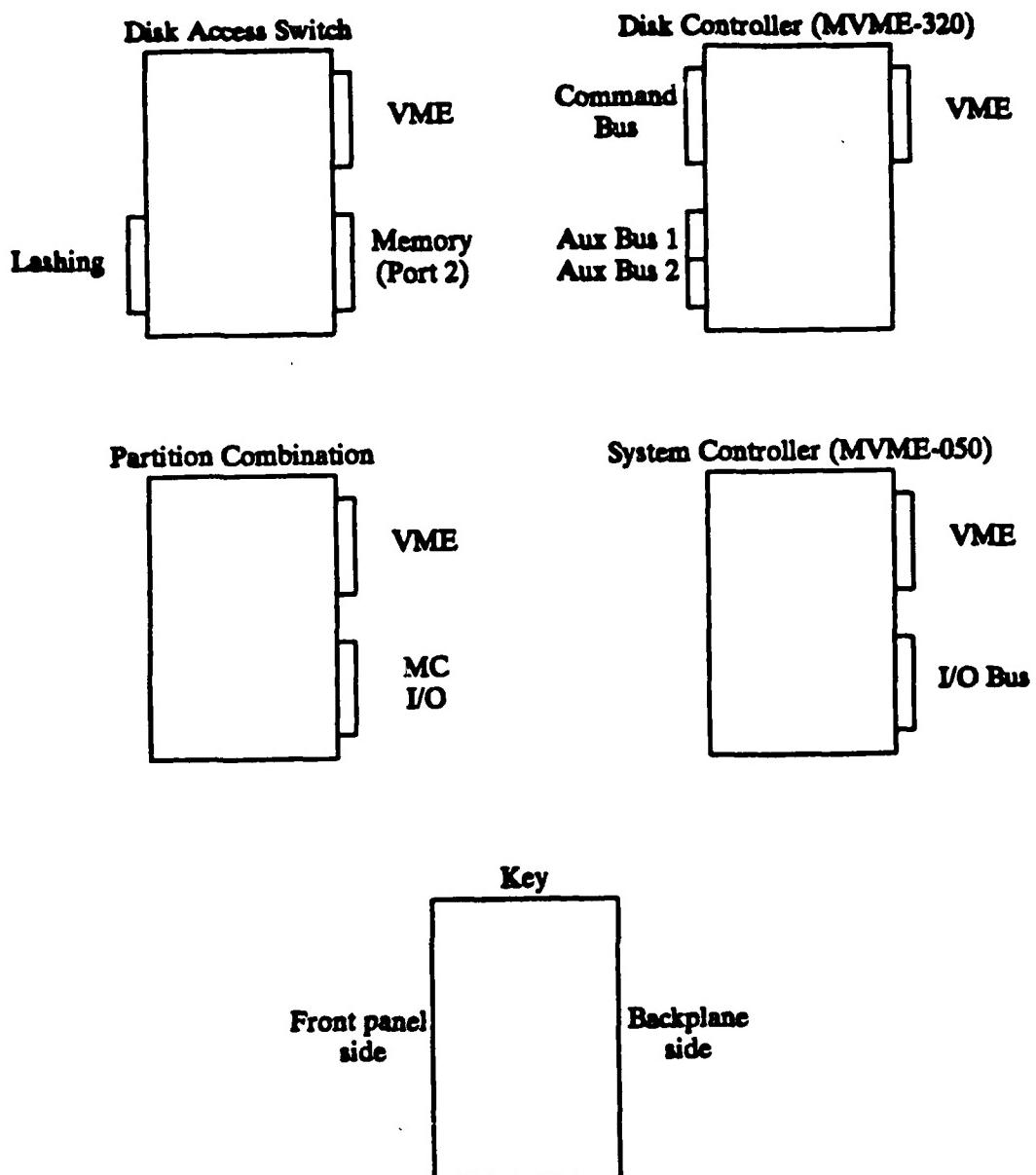


Figure 1.7.4 Disk Access Switch, Disk Controller, Partition Combination, and System Controller Boards.

serial port cannot. Address mapping or protection for VME bus addresses must be done off-board since there is no on-board area or scheme for a memory management unit. In the MVME-110, the "lower" connector implements a simple *I/O Channel Interface* that can be used to control a variety of peripheral devices. This interface is not used in the PEs however.

The use of the MC68000 or MC68010 poses a potential problem for SIMD mode operation. The difficulties arise when an exception occurs (interrupts are masked, so they cannot occur). Before beginning exception processing, the contents of the CPU's internal *instruction prefetch buffer* are discarded and replaced with instructions of the exception handler. Since the instructions in the buffer were SIMD instructions, they can never be re-retrieved because the PE has already "taken" them. For many exceptions this is not a problem because there can be no reasonable recovery from the error, e.g., bus error, privilege violation. However, some exceptions occur normally, for example, a trap to a software simulation of a floating-point instruction or an operating system call. For these, the compiler must arrange the PE instruction stream such that the contents of the prefetch buffer may be safely thrown away. Since the MC68000 has a 1-word prefetch and the MC68010 has two words, either one or two "no-operation" instructions following any instruction that can cause a "legal" exception are needed.

Each of the Memory Boards (Figure 1.7.1) is an arbitrated-access dual-ported memory. Since two boards are being used, so long as the CPU accesses memory on one board and the disk unit the other, there are no conflicts and no arbitration is done. When the CPU and disk unit attempt to access the same memory, they get alternating memory cycles. The CPU accesses the memory through the upper backplane connector (port 1); the MSU disk through the lower (port 2).

Four banks of nine sockets are provided on the Memory Board for dynamic memory chips. The chip occupying the ninth socket in each bank stores a parity bit associated with each byte. If 64K-by-1 chips are used, the memory capacity per board is 256K bytes; 256K-by-1 chips give a capacity of 1M bytes; and 1M-by-1 chips yield a total of 4M bytes per board.

No memory management functions are done on the Memory Board. Since there is no protection of "pages" of memory, user programs cannot be protected

from each other and operating systems programs must completely fit within the CPU memory (or be subject to failure by being corrupted). The former problem is quite serious because it allows an error in one user's program to destroy data being loaded/unloaded into/from the other memory unit. Further, such an error cannot be detected. This inhibits the usefulness of the double-buffered memory concert if correct operation is to be guaranteed. Another loss is that no part of the PE memory can be safely reserved as a PE-MSU message exchange area because of its vulnerability. The lack of memory mapping or even a segment base register further limits the usefulness of the PEs. Since the operating system cannot place programs and data where it wants, either programs must be relocated by a linker-loader just before execution or else programs must be compiled and loaded such that they all start in the same place in physical memory. The former solution is rather inefficient because there is potentially a very large number of relocatable addresses in a program. The later solution is no better in that only one program can reside and be operated upon in PE memory at once. Pre-loading the double-buffered memory with the "next" program would not prove to be very useful since that program will have to be moved in memory before it is executed. Fortunately, data can be pre-loaded without penalty. No support for trapping shared memory addresses is provided on the Memory Board either. Both memory mapping and protection were traded away to meet development time deadlines and to aid the layout, debugging, and test processes. Although several commercially-produced dual-ported memories are available, they did not meet budgetary constraints.

The MC-PE I/O board (Figure 1.7.1) handles instruction fetching in SIMD mode. It generates requests for SIMD instructions and provides them to the PE CPU over the VME bus when addressed. Because the MC68000 uses an asynchronous bus protocol, the bus timeout is disabled for SIMD instruction fetches. In retrospect, the SIMD instruction fetching performed on the MC-PE I/O board would be better if it returned a "no-operation" instruction to the PE when no instruction from the MC is available. The current scheme allows the MC to get the attention of a PE in SIMD mode only by writing instructions to it via the Instruction Broadcast Register. This might not be desirable in some error handling situations.

Also on the MC-PE I/O board are a *Condition Code Register* and its related *Combining Logic* which are used to send a 1-bit condition code bit to the MC to be used in forming a data conditional mask. A CPU chooses the bit to send by performing two actions: selecting a boolean Combining Logic function to perform on the condition codes and then writing the condition codes to the Condition Code Register. The Combining Logic functions are the same sixteen boolean functions that are used by the MC68000 conditional branch instructions (e.g., non-zero, greater than, overflow, less than or equal). Condition codes themselves are obtained by writing the lower byte of the CPU status register (the condition code byte) to the external Condition Code Register. Four of the bits are significant to the Combining Logic: they are the carry, overflow, negative, and zero flag bits. The Condition Code Register is accessed and the Combining Logic is controlled via the VME bus.

Two MC68230 Parallel I/O and Timer chips are also found on the MC-PE I/O Board. Each MC68230 has two 8-bit general-purpose I/O ports (A and B), a special-function 8-bit port (C), and a 24-bit counter. These chips were primarily added as performance monitors to count events of various types (e.g., bus read cycles, bus write cycles, accesses to the SIMD Instruction Space). The system clock can be counted and used as a "time base" to determine parameters such as bus utilization, percentage of accesses for SIMD instructions, and so on. Which events to monitor are software-selectable by setting I/O bits of the A and B ports in each of the MC68230 chips. Also, these chips can act as general purpose interval timers for benchmarking and analysis of the execution of programs. The timers are controlled via the VME bus.

A socket for the Motorola MC68881 Floating Point Co-processor [Mot85] is also provided on the MC-PE I/O Board. While the MC68881 is primarily intended for use as a co-processor with a more advanced member of the 68000 family, the MC68020 [Mot84b], it can be used as a peripheral to any CPU. This floating point processor, although currently many times the cost of an MC68000 CPU, can significantly increase the processing rate of programs using floating point arithmetic. Without it, floating point instructions must be simulated in software. The MC68881 is controlled via the VME bus.

Finally, the MC-PE I/O Board has a set of chips implementing the General Purpose Interface Bus (GPIB) standard. PEs use the bus to interrupt and

communicate with their MCs. MCs can also interrupt the PEs over this channel. There is no hardware provision for detection of lost or corrupted messages sent over the GPIB bus. The GPIB bus controllers are accessed via the VME bus.

The PE-Network Interface Board (Figure 1.7.1) consists of two MC68230 parallel I/O ports, a data transfer watchdog timer, parity encoder and decoder, and checking logic to ensure correct parity and path routing. One half of each of the two MC68230s' ports (A and B) is used as DTRin, the other two halves are used for DTRout. This organization was chosen because each MC68230 has only an 8-bit data bus connection; therefore, always using them as a pair allows the CPU to read/write 16-bit data from/to the network in one bus cycle. Two parity bits accompany each 16-bit word passing through the network, one for each byte.

A CPU requests a path through the circuit-switched Extra Stage Cube network by first writing a 16-bit word consisting of the destination and broadcast tags to the DTRin. The network does not begin to establish the path until the CPU sets the "Path Request" bit on port C. Because the MC68230 also has an interval timer, it is set by the CPU to an appropriate value and used as a path request watchdog timer. The CPU then polls the "Path Grant" bit on port C, waiting for it to indicate that the path has been established. If the path is granted, the CPU cancels the path request watchdog timer; otherwise the timer expires and the CPU enters interrupt processing. At this point, the CPU can give up or try to establish the path again.

At the destination end of the network, the PE-Network Interface Board checks the incoming "path request" for good parity and also compares the destination tag against the destination PE's number. If all is well, the path request signal is reflected back to the source as the "path grant." This checking is done on the PE-Network Interface Board and does not require the intervention of the destination PE. Any discrepancy during the checks for good parity are routing result in a "path grant" never being generated causing the source's path request watchdog timer to expire.

Once the path is established correctly, unidirectional 16-bit data transfers can begin. The DTRs engage in an interlocked handshake protocol as described in the last chapter. Each time a data item is output from DTRin

into the network, the hardware data watchdog timer is enabled. This is done automatically so that the CPU does not have to explicitly set a timer each time it transfers a word through the network. The timeout value of the data watchdog timer is software-controllable to accommodate network faults or the presence of extra stage delays. As each data item arrives at the destination, parity is checked. If the parity is incorrect, no acknowledgement is sent back to the source and the data watchdog timer will expire. At this point, the CPU can give up, try to send the data again, or even try re-establish the path and restart the transfer.

When the transfer is complete, the source is responsible for dropping the path by writing the "Path Drop" bit in port C. It is a grave error to drop the path while data exists in the internal output buffers of the DTRs or is being transferred through the network.

Two other port C bits are used to control whether the input (extra) stage and the output stage are bypassed or not. Writing a final port C bit places the network in a "fault diagnosis" state during which bad parity and routing conditions are reported to the destination PE as well as the source. This aids in the identification and location of network faults. To provide the fault-tolerance, the DTRin output and DTRout input are split to provide both the "normal" and the "bypass" connections to the Extra Stage Cube network. Unfortunately, the normal and bypass signals leave the board through a single connector as shown in Figure 1.7.2.

Another single-point failure that can occur is a fault in the "Path Request" or "Path Grant" signals. These are not duplicated either in the cabling nor in the network itself. Thus a single "stuck-at" path request signal in one stage of the network will be propagated through all of the remaining stages. Even though the input and output stage bypass controls are not replicated, they are not a single-point failures since the state that they are causing can be deduced and accommodated. The PE-Network Interface Board is accessed via the VME bus.

A five-card-wide VME backplane connects all of the boards of a given PE. Also, the lower connectors of the MC-PE I/O boards of PEs in an MC-group are tied together and connected to the MC. These lines carry the PE instructions broadcast by the MC, enable signals, and GPIB bus signals.

PEs lack an NIU. This implies that all shared memory requests must be handled in software. The network is unidirectional; therefore, shared memory requests will be rather costly. While the interlocked handshake and parity checking of the network interface prevents loss or damage to messages, handling network re-tries is the direct responsibility of the CPU. Network fault diagnosis also requires explicit CPU support. No DMA capability has been implemented either. Thus no local-remote memory copy operations via the network can be done without explicit CPU instructions. All of these "omissions" are the result of limited budget and development time.

The set of PE boards can be used in any PASM system up to $N=128$ and $N/Q=8$. The board limiting N is the PE-Network Interface because the $(n+1)$ -bit destination and broadcast tags must be packed into a 16-bit word. Since the GPIB bus limits the number of talkers/listeners to be sixteen, no more than eight PEs and the MC may share it.

Interconnection Network Design

The interconnection network is an Extra Stage Cube with $n+1$ (five) stages. It is made up of three types of boards: Input Stage Boards, Intermediate Stage Boards, and Output Stage Boards (Figure 1.7.2). There are $N/2$ (eight) boards of the same type in each stage, each board implementing a 2-by-2 crossbar switch. As a result of network simulation studies, it was determined that the ordering of the stages should be Input (cube_3), Intermediate (cube_0), Intermediate (cube_1), Intermediate (cube_2), and Output (cube_3) for the best performance during the path establishment phase [DaS85b, DaO85].

Each board type centers around a synchronous finite-state machine which controls four banks of line drivers using four signals, C_{1-4} (Figure 1.7.5). For example, to perform the "straight" connection, C_1 and C_4 are asserted; the lower broadcast is performed by asserting C_3 and C_4 . The finite state machine changes state during path establishment and path drop operations. It also controls the combination of destination-to-source handshake signals that result from broadcast connections. The Input and Output Stage Boards have additional circuitry to implement the "bypass" connections.

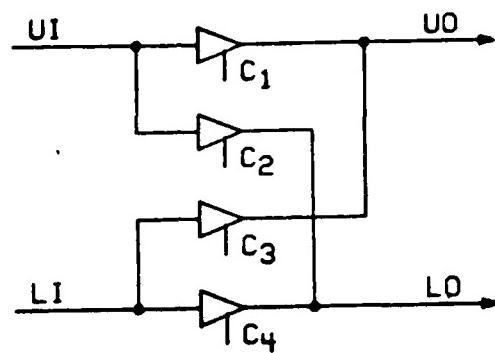


Figure 1.7.5. 2-by-2 crossbar switch controlled by signals C_{1-4} . UI and UO are the upper input and output, respectively. Similarly, LI and LO are the lower input and output.

The Input Stage Board accepts signals from two different PE-Network Interface boards at two front-panel connectors. Two sets of data and handshake signals from the same PE appear at each connector: one for the "normal" path and one for the "bypass" path. Signals that control the network such as "Path Request," "Path Grant," and the normal/bypass selection are not replicated on a connector. The Output Stage Board is similar in that it also has two front-panel connectors that carry signals to two different PE-Network Interface boards. All communication between network boards occurs via rear-panel connectors and parallel flat cabling.

MC Design

A prototype MC consists of six boards: a CPU Board, two Memory Boards, an MC-PE I/O Board, a Fetch Unit Board, and two Parallel Port Boards. Of these, only the CPU Board is a commercial product.

The MC is organized in a "Master MC CPU - Master/slave FBU" configuration. Characteristics of the CPU Board, Memory Boards, and the MC-PE I/O Board were described earlier.

Each of the Memory Boards (Figure 1.7.1) is the arbitrated-access dual-ported memory described earlier. The CPU accesses the memory through the upper backplane connector (port 1); the Control Storage disk through the lower (port 2). Comments about the lack of memory management, protection, cache, and other features that were made in the PE description section can also be applied to the MC design and will not be repeated here. No reconfigurable MC-MC memory bus scheme (Figure 1.6.7) nor a pairwise MC-MC memory connection has been implemented in the prototype. Such a scheme would have required the design of at least two additional board types; one to interface the VME bus Memory Board(s) to the MC and the other to interface the FBU Memory (not a VME bus) to the FBU. Sharing the MC memory space among two or more MCs would have also complicated the software that each MC uses to manage its own memory space.

While the MC-PE I/O Board is used in both the MC and PEs, it is populated with different chips depending on its location. In the MCs, this board would use an extra chip that acts as a GPIB controller. Since the GPIB

requires just one of these chips per bus, it seemed logical to situate it in the MC. MCs probably would not typically need to perform floating point operations; therefore, it is unlikely that the MC68881 Floating Point Co-Processor will be installed on MC-PE I/O Boards in PEs. Further, MCs do not need the Condition Code Register and Combining Logic on their boards; however, they do need an N/Q-bit Data Conditional Mask Register to accept the condition codes from the PEs. An 8-bit register is used for this purpose, four bits for the condition codes themselves and a "presence" bit that indicates when each of the PEs has written its condition codes. The current implementation of the board limits N/Q to be four due to the size of this register. A slight re-design could remove this restriction, leaving the N/Q=8 limit due to the GPIB established earlier. Finally, a 4-bit register appears on the MC-PE I/O board that is used by an MC to contribute its local "if any"-type condition and synchronization signal to the "MC status tree" and to retrieve the global result and synchronization information.

The Fetch Unit Board (Figure 1.7.3) consists of an FBU and 32K bytes of fast static FBU Memory. As detailed earlier, the FBU Memory contains only PE instructions for SIMD programs. Like the dynamic Memory Boards, memory access is arbitrated between the FBU and disk "ports." The FBU feeds a PE Instruction Queue which holds 64 16-bit instruction words and the 4-bit enable signals that accompany each of them. The enable signal width limits the use of this board to systems where N/Q=4. A 16-bit-word Immediate Broadcast Register and a 4-bit Enable Signal Register are writable from the MC CPU.

Internally, the FBU consists of a finite-state machine that controls the fetching of the PE instructions from FBU memory and the filling of the PE Instruction Queue. It is controlled by the MC CPU via three registers: BASE, START, and COUNT. BASE acts as a segment base register; therefore, programs can be freely relocated in FBU Memory. The starting FPC is calculated by the FBU by adding BASE and START. COUNT is an 8-bit register that holds the number of 16-bit words that should be fetched. Because 68000 instructions are always an integral number of 16-bit words in length, COUNT gives a word rather than byte count; therefore, up to 255 words can be fetched and broadcast using one START-COUNT setting. For efficiency reasons, the

START and COUNT registers are arranged in memory such that they can be written by a single 68000 instruction.

One problem with the current FBU design is that the MC CPU is prevented from writing BASE, START, or COUNT while the finite-state machine is operating on the current "work" specified by the registers' contents. Of course, this is what is desired. The problem is that delaying the write also prevents an acknowledge signal from being returned to the MC CPU for an unspecified time, i.e., due to the PEs not emptying the queue, and causes interrupts to be ignored for the duration. This was discussed in an earlier chapter when this MC organization was presented.

Here some approaches to prevent the MC CPU from being blocked are presented. One approach would be to immediately cause the bus cycle to be terminated by having the FBU generate a bus error if it is busy. The MC CPU could check if it was trying to write the FBU and if so, try again later. The current design generates a bus error but only after a bus watchdog timer expires. This prevents this waiting time from being used more productively and locks out interrupts as discussed earlier. Another approach to the problem would be to add an "FBU work queue" that would automatically interrupt the MC CPU when a "high-water-mark" of work was exceeded. Another interrupt could be generated at a "low-water-mark" to induce hysteresis. Both of these schemes prevent the MC CPU from blocking.

The Parallel Port Boards (Figure 1.7.3) are used for general-purpose communication between the MCs and the SCU, the MCs and Control Storage, and between the MCs and the Memory Management System. A Parallel Port Board has two MC68230 parallel ports and two front-panel connectors. One half of each MC68230 port is set up for 8-bit unidirectional output; the other half for input. Therefore, one board can implement two bi-directional parallel channels. CPUs access the Parallel Port Board via the I/O Channel Interface.

The first port on the first Parallel Port Board is connected to the SCU; the other port is connected to the Directory Processor in the Memory Management System. One of the ports on the second board is used for direct communication to Control Storage.

MCs lack a network to communicate among themselves (other than the 1-bit status tree). This is not a particularly serious problem but probably

restricts the range of control and synchronization operations the MCs can do efficiently. For example, semaphore handling in MIMD mode will either have to be carried out entirely at the PE level or else will involve the SCU to a greater extent than necessary. Lack of a DMA controller is also a loss, although not as great a one as for the PEs. Although the MCs do not have a PE address mask decoder, the number of unique PE address masks that can be specified for an N-PE system is given by 2^{2n} . For N=16, the number of masks is only 256; therefore, the 16-bit general masks associated with each PE address mask can easily be stored in MC memory and accessed by a single table-lookup operation. This scheme is most appropriate for machine sizes up to and including N=64.

The boards comprising an MC communicate via a six-slot VME backplane with the exception of the Parallel Port Boards that use the I/O Channel Interface. The MC MC-PE I/O Board and Fetch Unit Board have their connectors tied to each of the PE MC-PE I/O Boards for instruction broadcast and GPIB communication.

Control Storage and MSU Design

Both Control Storage and an MSU consist of five board types: a CPU Board, a Disk Controller Board, two Memory Boards, a Disk Access Switch Board for each memory that the disk serves, and Parallel Port Boards for each device that can request service from the disk. Of these, both the CPU Board and Disk Controller Board are commercial products. The disks themselves are Maxtor 85M-byte Winchester-technology devices. Only one disk unit per MSU or Control Storage is used.

The CPU Board, Memory Boards, and Parallel Port Boards have been previously described. Control Storage has five Disk Access Switch Boards, one for each of the MC memories and one for the SCU memory. Each of the MSUs has four Disk Access Switch Boards, one for each of the PE memories it serves. Requests for Control Storage disk service come from any of the MCs, the SCU, or the Input/Output Processor (part of the Memory Management System). Therefore, six Parallel Port Board channels (three boards) are needed. MSUs need only two channels since they receive commands from either the Command

Distribution Processor (part of the Memory Management System) or the Input/Output Processor.

The Disk Controller Board is the Motorola MVME-320 (Figure 1.7.4) which can control a variety of Winchester-technology (hard disk) and floppy disk units. It can control up to four disk units at a time, two of them Winchesters. although in the present application, it will control just one. As more disk space is needed, disk units can be added without the need to change controllers. The MVME-320 has an on-board CPU (non-68000-family) and a DMA controller. The controller is programmed by the MC68000 CPU and transfers its data to the Memory Boards via the VME bus.

The Disk Access Switch Board (Figure 1.7.4) is designed to interface the second port of the dual-ported memories to a VME bus. Consider the operation of loading a PE memory from an MSU. First, the MSU CPU sets the Disk Access Switch Board to connect it to one of the two PE memory units (recall there are two PE memory boards, each of which has a "port 2"). Next, the data to be loaded is retrieved from the disk unit by instructing the Disk Controller to fetch one or more disk blocks and to store them "through" the switch, thus avoiding the MSU memory. For output from a PE memory, the Disk Controller can be instructed to fetch from the PE memory through the switch, also avoiding the MSU memory.

Due to the comparatively long times needed to read/write the disk media (compared to primary memory access) it may sometimes be desirable for the MSU CPU to buffer disk blocks in its own memory temporarily before they are written to the PE memory or after they are read from the PE memory. This is especially true when processed data needs to be unloaded from the PE memories before data for the next task can be loaded. If the unloaded data had to be written onto the disk immediately, this might interfere with the disk accesses needed to retrieve the data for the next task. If a memory scheduling policy calls for anticipatory prefetches, the MSU memory would also be used to temporarily store the disk blocks that may be needed. When the blocks are demanded, they can then be read from the MSU memory and stored through the switch much faster than if the access required a disk read operation.

When several PEs are to be loaded with the same data, it would be inefficient to fetch and store to each of them individually. The Disk Access Switch Board has been designed to allow two or more of the boards to be "lashed together" so that when the MSU CPU stores through one switch, the data is broadcasted to all of the selected PE memory units simultaneously. The current board design limits the number of boards that can be "lashed" to eight; therefore, a machine size which has N/Q greater than eight cannot take advantage of loading the PEs in one step.

The Disk Access Switch Boards lack DMA capability. This would be of great use to allow the MSU or Control Storage CPU to perform other tasks such as file system operations while the data loading/unloading operations are being performed. In this application, the DMA controller would be made to be programmable via the MSU VME bus, but would control the transfer of data between port 2 of the MSU memory and port 2 of the PEs. This would prevent the MSU to PE transfer from interfering with the disk unit to MSU Memory transfer.

System Control Unit Design

The prototype SCU consists of a CPU Board, two Memory Boards, a Partition Combination Board, a System Controller Board (MVME-050), an Ethernet Interface Board, and several Parallel Port Boards. The CPU Board, Memory Boards, and Parallel Port Boards have been described earlier. The SCU needs parallel ports to communicate with each of the MCs, Control Storage, the Directory Processor, and the Input/Output Processor for a total of seven channels (four boards).

The SCU for the prototype is responsible for the overall orchestration of the activities of the system. It shares its mass storage device, Control Storage, with the MCs. In order to allow a larger group of users to access PASM, the SCU will serve as a link between PASM and the Engineering Computer Network (ECN). ECN is a local network of 20-30 computers of varying capability at Purdue University. A PASM user's terminal will be physically connected to an ECN host computer. The host will provide the environment for the development, compilation, and debugging of SIMD and MIMD programs to

prevent the SCU microprocessor from being burdened. Commands (jobs) initiated by users are sent by the host to the SCU, which schedules the jobs to be run on the parallel machine. ECN communication will be accomplished via the Ethernet Interface Board.

The Partition Combination Board (Figure 1.7.4) controls the combination of a variety of signals for partitions consisting of more than one MC-group. The SCU programs the combination logic to indicate the assignment of MC-groups into partitions. To implement the global "if-any" conditional calculation, each MC sends a 1-bit local condition and a 1-bit local synchronization signal to this board and each receives a 1-bit global condition and a 1-bit global synchronization signal in return. Similarly, to ensure that all of the PEs in a multiple MC-group partition request an instruction before any MC releases one, each MC contributes its local combined PE instruction request to this board and receives a global PE instruction request appropriate to the partition.

The Partition Combination Board also distributes reset signals to the MCs used during boot-up procedures. In addition, the timing of the global refresh of dynamic memory is controlled by this board. Refresh is done by all Memory Boards in the system simultaneously. Failure to do so would allow PEs and MCs to cause each other to have to wait during refresh cycles. This is clearly undesirable behavior for large systems in which there is a high likelihood that *some* Memory Board would need refreshing.

The SCU is a convenient place to locate system peripherals such as a time-of-day clock, printer, the system console, system status displays, boot-up floppy disk drive, backup tape drive, and so on. The System Controller Board simplifies interfacing many of these functions to the SCU.

Memory Management System Design

The Memory Management System is the least-well-defined part of the PASM prototype, mainly because the number of processors needed and the best way to arrange them will depend on how the operating system is written. Currently, plans are to place three of the Memory Management System processors, the Directory Processor, Memory Scheduling Processor, and Command Distribution Processor on the same VME bus together with one or more

Memory Boards. The shared memory space provided by the Memory Boards would be used to store operating system tables that are needed by all three processors. In order to reduce contention to the shared memory, the three processors would execute programs stored locally in the CPU Board EPROM area and store local data in the CPU Board RAM. Also to reduce bus contention, inter-CPU communication would be via parallel ports. Recall that the Parallel Port Boards use the I/O Channel Interface and not the VME bus.

As shown in Figure 1.4.2, the Directory Processor communicates with each of the MCs, the SCU, and with the Memory Scheduling Processor. Therefore, six parallel port channels are required. The Memory Scheduling Processor needs only two connections: one with the Directory Processor and one with the Command Distribution Processor. The Command Distribution Processor controls the four MSUs and also communicates with the Memory Scheduling Processor and the Input/Output Processor for a total of six parallel port channels.

The last processor involved in memory management is the Input/Output Processor. Its role is to be in distributing data arriving from external sources among the MSUs and Control Storage. It may have a second Ethernet port with ECN so that I/O devices on other ECN machines can send or receive data to/from PASM without bothering the SCU.

The Input/Output Processor is to consist of a CPU Board, two Memory Boards, five Disk Access Switch Boards, and a number of Parallel Port Boards. The Disk Access Switch Boards allow the processor to read or write data into the second port of the Control Storage and MSU memories. A transfer of data from the "world" external to PASM and to be distributed among the MSUs might proceed as follows. Data is first retrieved from an external source by the Input/Output Processor and placed in its memory. From there, the processor can decide how it is to be distributed among the MSUs (e.g., each gets a copy, each gets every N/Q'th "row" of data). Next the Input/Output Processor sets the Disk Access Switch Boards to connect them to the correct MSU memories. After consulting with the MSU processors about where the data should be written, the transfer can be performed.

The Input/Output Processor is functionally similar to the actions of MPP's staging memory in that a single stream of data can be distributed among the array of PEs. However, MPP's staging memory is much larger and

faster than the Input/Output Processor and can process and reformat entire vectors of data simultaneously: the Input/Output Processor is a word-at-a-time device. MPP's staging memory can also handle real-time data input and output; the Input/Output Processor cannot. The staging memory achieves its speed because it is special-purpose hardware that performs the coordinated movements of data between internal memories. While it allows some rather sophisticated byte-stream transformations, it is not completely flexible like the Input/Output Processor. For example, it cannot perform any data-dependent operations such as clipping or thresholding.

Real-time I/O is an acknowledged problem for the PASM prototype. The PE memories are at least two steps away from the I/O device: data must first be read by the Input/Output Processor, then written to the MSU Memory, and finally written into the PE Memory. Ideally, a set of Disk Access Switch Boards fed by a high-speed switch distributing the real-time I/O device's data would be used to eliminate the "middlemen" MSU and Input/Output Processors.

CHAPTER 8

LESSONS FROM THE PROTOTYPE

This chapter bridges the gap between the previous summary of the PASM prototype and the proposal of a design and implementation for a 1024-PE PASM system. It draws on the experience of designing and building the PASM prototype and is a "self-criticism" of its implementation scalability.

The number of physical boards required to build the PASM prototype is summarized in Table 1.8.1. For comparison, the number of boards required to build a PASM system of 1024 PEs and 32 MCs using the same technology is given. Clearly, the implementation scalability of the PEs and interconnection network is poor. The MC and MSU implementations are less critical, but improvements need to be sought.

There are clearly too many boards per PE in the prototype for implementation scalability. Some of the problem areas in the prototype PE implementation will now be discussed.

The high board count is due in part to the double Eurocard VME bus standard which calls for a rather small board area. Approximately 10-20 percent of each board's area is dedicated to chips associated with the VME bus interface, the I/O Channel Interface (which is not used at all in the PEs), or the sockets used to interface the board with the backplane. A considerable amount of space could be saved by not replicating these items as would be possible if a single-board PE were designed. Simply using the CPU bus directly rather than the VME bus would eliminate bus drivers that currently waste board area, consume power, and add to propagation delays.

Another problem is the path width of the interconnection network: sixteen data bits, two parity bits, and several control lines are required for both the paths from the PE to the input of the network and from the output of the

Table 1.8.1. Physical board requirements using prototype implementation technology.

| COMPONENT | GENERAL EXPRESSION | PROTOTYPE IMPLEMENTATION | FULL PASM IMPLEMENTATION |
|-----------|-----------------------------|--------------------------|--------------------------|
| PE | 5N | 80 | 5120 |
| NETWORK | $\frac{N}{2}(\log_2 N + 1)$ | 40 | 5632 |
| MC | 7Q | 28 | 224 |
| MSU | $9\frac{N}{Q}$ | 36 | 288 |

network back to the PE. The situation is further exacerbated by an additional fully redundant path between the input and output of the network and the PE DTRs. This results in a complete board being used for this PE-network interface.

Separate data paths are used between the MC and the PEs for broadcasting instructions and for the asynchronous MC-PE communication. Further, the MC-PE communication uses parallel data transfers. A great deal of board area and connector space is dedicated to this communication. Clearly, instruction broadcasts should be carried on parallel lines, but the low-traffic MC-PE communication channel is probably not worthy of fully-parallel connections.

The double-buffered dynamic memory is complicated by the need to arbitrate the asynchronous requests arriving from the CPU and the disk port and by the need to refresh the dynamic memory. The separate 16-bit data and 23-bit address paths between the MSU and PE are also taking up too much board area. The memory lacks error detection and correction (ECC) logic to allow recovery from single memory errors. Such errors will become more likely as the total amount of memory becomes larger as would be the case in a 1024-PE PASM implementation.

The other component for which the prototype implementation is not scalable is the interconnection network. Each Network Interchange Box board implements a 2-by-2 crossbar that is sixteen data bits, two parity bits, and several control bits in width. A substantial portion of the board is dedicated to lines rather than chips. Not enough experience has been gained to determine if the redundant PE-Network connections actually enhance the overall system reliability.

Board count reductions in the MCs and MSUs are less critically needed, but will result automatically if appropriate steps are taken in the PEs to reduce data path width and to place CPU, memory, and communications on the same board. A high-speed I/O channel allowing direct access to the PE memories is highly desirable for real-time processing.

The next chapter explores some of the tradeoffs to be considered for a 1024-PE PASM system.

CHAPTER 9

TOWARD A 1024-PE PASM SYSTEM

In this chapter, specific performance goals are outlined for a 1024-PE PASM system and the hardware required to achieve them is detailed. Special attention is given to physical system layout and interconnection.

1.9.1 Performance Goals

Because image processing is one of the primary forces driving the PASM "design philosophy," some performance goals should be set before arbitrarily deciding that 1024 PEs are "enough." One goal should be the processing of 1K-by-1K monochrome images (one byte per pixel) at TV rates (30 images/second). The processing should be a significant enhancement algorithm such as contrast enhancement or edge sharpening. The performance of such algorithms is dominated by the time spent in fetching pixel values from memory. Thus by counting the number of times each pixel is fetched, a rough estimate of the number of instructions the machine must execute per second can be derived. Suppose that a good edge sharpening algorithm uses 5-by-5 pixel "windows" which must be compared at each "match position" in the image to enhance the edge. Each pixel would then be fetched up to 25 times, once for each "match position." If 20 machine cycles are required to fetch each pixel (appropriate for 68000-family processors, including requisite index calculations), 500 machine cycles are expended on processing each pixel. Since 1K-by-1K times 30 pixels must be processed per second (30M pixels/sec), the machine must provide 15 Giga-cycles per second. Current microprocessors in the 68000 family have 12.5 to 16.67 MHz clocks, making 1024 processors sufficient to achieve this processing rate. If more complex algorithms are to be

executed, the number of PEs or speed of the CPUs can be increased or the processing rate can be reduced.

Another goal determines the desired memory size for each PE. Satellite images approaching 5000-by-5000 pixels are now being considered by those involved in digital photogrammetry. Often these images have data from several sensors, resulting in storage requirements of several bytes per pixel. Assuming four bytes/pixel, 100M bytes of memory are required. If each of the 1024 PE memory units had a capacity of 128K bytes and the image were distributed evenly among the PEs, five bytes per pixel could be stored in each PE. This could be the entire image as well as a "working image" of one byte per pixel.

In order to compare the memory requirements of the PASM prototype and 1024-PE PASM system, a fixed *subimage* size (image size per PE) is assumed. Therefore, the minimum memory requirement for a prototype PE would also be 128K bytes. Because the prototype is being used as a research system and is required to store and process large images, significantly larger memories than 128K bytes are being used. The prototype does not have sufficient memory to hold 5K-by-5K images in memory; however, 1K-by-1K images can be stored and processed, although not as fast as a 1024-PE system.

1.9.2 Hardware Constraints

Frequently in image processing algorithms, a PE must transfer its subimage edge pixels to neighboring PEs. As subimages become smaller, the use of DMA for the "block transfers" becomes less efficient due to setup overhead of the DMA controller. Because of this, the interconnection network also experiences more short bursts of traffic. Packet switching has been determined to be a desirable alternative to circuit switching under these conditions. Also, the probability of network contention increases for circuit switching as network stages are added making it less desirable for very large numbers of processors.

In scaling up the number of PEs to 1024, the number of MCs was set at 32. This decision is made for practical reasons as 32 represents reasonable limit on the number of PEs that could be placed close enough together to be interconnected by a bus.

Another constraint is that in scaling up to 32 MCs, the number of MSUs increased to 32 and the number of PEs serviced by an MSU became 32. Assuming a fixed subimage size, the amount of data handled by each MSU has increased eightfold. Of more concern is the eightfold increase in overhead for servicing the load/unload requests. This suggests that 32 MSUs might not be enough when the machine is operating in MIMD mode (in SIMD mode, the disk volume can be formatted such that there is no difference in the overhead). More experience with the prototype will be necessary before determining if an MSU servicing four PEs is about right or whether it represents excess capability.

Using the previous discussion of the limitations of the prototype and the design constraints given above, the design of a 1024-PE PASM system will now be outlined. LSI (1000 to 100,000 transistors) and VLSI (more than 100,000 transistors) approaches will be used to reduce the chip count, allowing more functions to be packed on each physical board. Use of LSI and VLSI chips will also reduce power consumption and increase reliability by internalizing interconnections. The use of commercial LSI and VLSI building blocks wherever possible because these are the result of hundreds or thousands of man-years of effort which cannot be duplicated except at extreme cost. The functions peculiar to parallel processing that are required for this design are to be implemented by components external to the commercial building blocks. It is not proposed to reduce chip counts by integrating these specialized functions into existing LSI and VLSI building blocks, even though it is feasible from a technological standpoint, because of high cost and the unwillingness of semiconductor manufacturers to share their high-performance designs. Rather, custom LSI approaches for the specialized functions are suggested where the need for space savings is greatest and where no commercial part is suitable.

1.9.3 1024-PE PASM Proposal

Figure 1.9.1 shows the proposed PE organization for such a system. It consists of a 16.67 MHz Motorola MC68020 CPU, a Motorola MC68881 Floating Point co-Processor (FPP), two dual-ported 128K-byte static memories with ECC logic, EPROM, a serial interconnection network interface, and a 32-bit parallel (plus control bits) interface to the MC. The MC68020 CPU has an

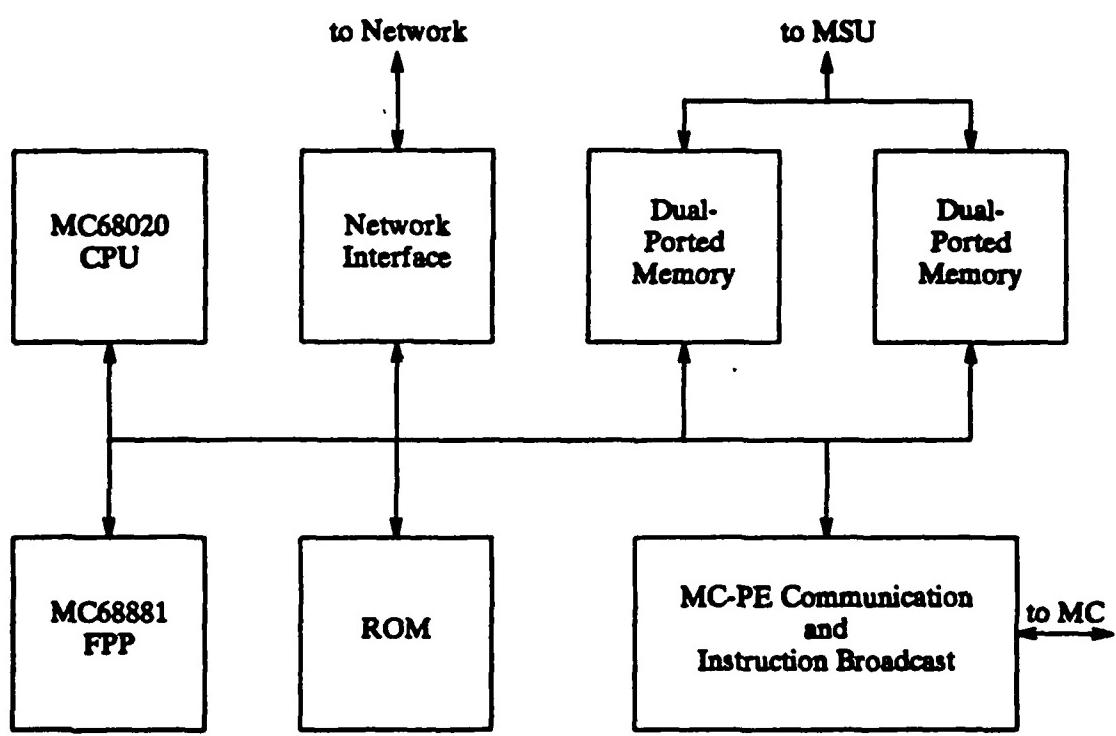


Figure 1.9.1. PE organization.

on-board instruction cache that enhances the PE's performance in MIMD mode. In SIMD mode, the PE gets its instructions from the MC through the MC interface; therefore the PE CPU's instruction cache is not used in SIMD mode. The MC interface allows bi-directional communication: in SIMD mode, instructions are broadcast to the PEs; in both SIMD and MIMD mode, PEs can request one-to-one communication with their MC to report their status or to alert an error condition.

The incorporation of the MC68020 CPU and MC68881 FPP in the design represents an appropriate use of commercial VLSI products. Even though there are some functions peculiar to parallelism that could be implemented in the CPU, a custom design would be unlikely to achieve the performance of these products. However, the serial interconnection network interface on the PE boards is an area where a custom LSI chip would be useful. It would construct data packets (this does *not* imply a packet-switching network), add parity or cyclic redundancy code bits to ensure correctness, and interact with the interconnection network. It could also handle the low-level aspects of a network protocol [Tan81a] such as character escaping, generation of acknowledgements and retransmit signals, and monitoring of "timeout" conditions.

A more intelligent DMA controller capable of handling certain types of data structures found in image processing is described in [KuS85]. Whereas the complexity of current DMA controllers is comparable to that of VLSI CPUs, it is not proposed to include a "super" DMA controller in the design for cost reasons.

Savings in board area as compared to the prototype implementation are due to the use of a smaller (physical and logical size) memory, static RAM chips (needing no refreshing hardware), simplified memory access control, multiplexed functions of the MC interface, a serial network interface, and reduced numbers of line drivers (no VME bus used). Use of grid-array and surface-mount technologies also markedly increase the density of the layout. Figure 1.9.2 shows the proposed PE board layout. The MC-PE interconnection bus connector mates with a horizontal backplane bus that interconnects the MC group. As shown in Figure 1.9.3, an MC group is arranged in a horizontal slice with 16 PEs on each side of the three-board implementation of the MC. This arrangement simplifies the driving of horizontal backplane buses and reduces

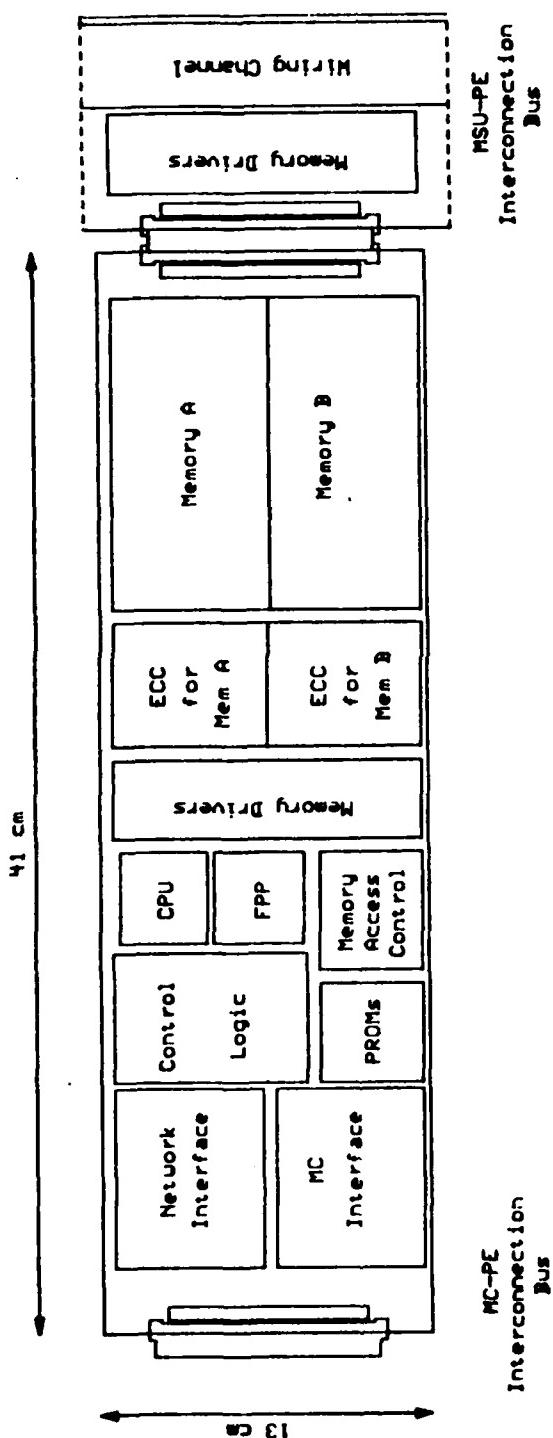


Figure 1.9.2. PE board layout for a 1024-PE PASM system.

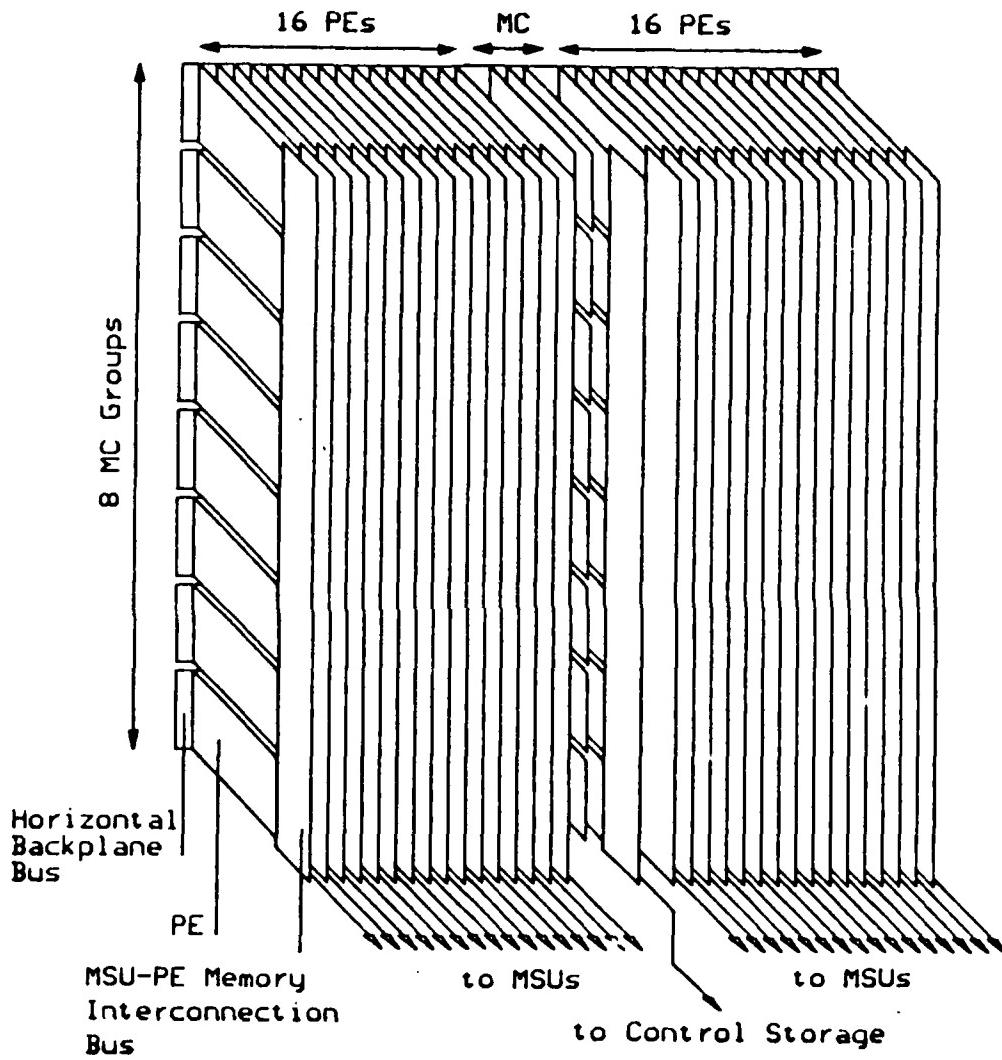


Figure 1.9.3. Physical organization of a PASM quadrant containing 256 PEs arranged in 8 MC groups.

propagation delays. Eight such MC groups are arranged vertically in a cabinet to form the quadrant shown in Figure 1.9.3. Four such quadrants comprise the set of MCs and PEs.

The logical PE i within each MC group is serviced by MSU i . Figure 1.9.3 shows that the MSUs are connected to the PEs through a vertical bus. The MSU addresses the memories of the PEs it services as a contiguous block; upper address bits select which PE's memory drivers are to be enabled for a data transfer.

The MSU organization is shown in Figure 1.9.4. Its organization is similar to that used in the prototype except that an External I/O Interface allowing connection to real-time I/O devices is added. Such an input channel to each MSU gives direct access to the PE memories by bypassing the secondary storage media. Use of the Buffered Interfaces (Figure 1.9.4) makes 4-way memory interleaving possible when accepting real-time data, enhancing throughput. With 32 MSUs each having the 4-way interleaved channels, there are 128 channels on which 32-bit data can be transferred. Assuming a 200 ns access time of the PE memories, an overall input speed to PASM in excess of 500 million 32-bit words per second can be achieved.

The most radical departure from the original prototype design occurs in the interconnection network. First, the 16-bit data path width in the prototype design which limited a physical VME-standard board to the implementation of a 2-by-2 interchange box could not be justified in a 1024-PE PASM system: over 5000 boards would have to be interconnected. Reducing the width to eight or four bits does not make enough of an impact on the number of boards required because at most a 4-by-4 interchange box could be fit on a single standard-sized board resulting in more than 1000 boards. The only practical solution seems to be a bit-serial network where the functions of a 2-by-2 or 4-by-4 interchange box would be integrated on one custom LSI chip.

When bit-serial networks are being considered, the choice between circuit-and packet-switching is not clear-cut. As described earlier, conflicts are more common, DMA is less-often used, and the delays due to establishing a network path and propagating acknowledgement handshake signals are increased for large circuit-switched networks. On the other hand, once a circuit has been established, the throughput of the network is higher than that for packet

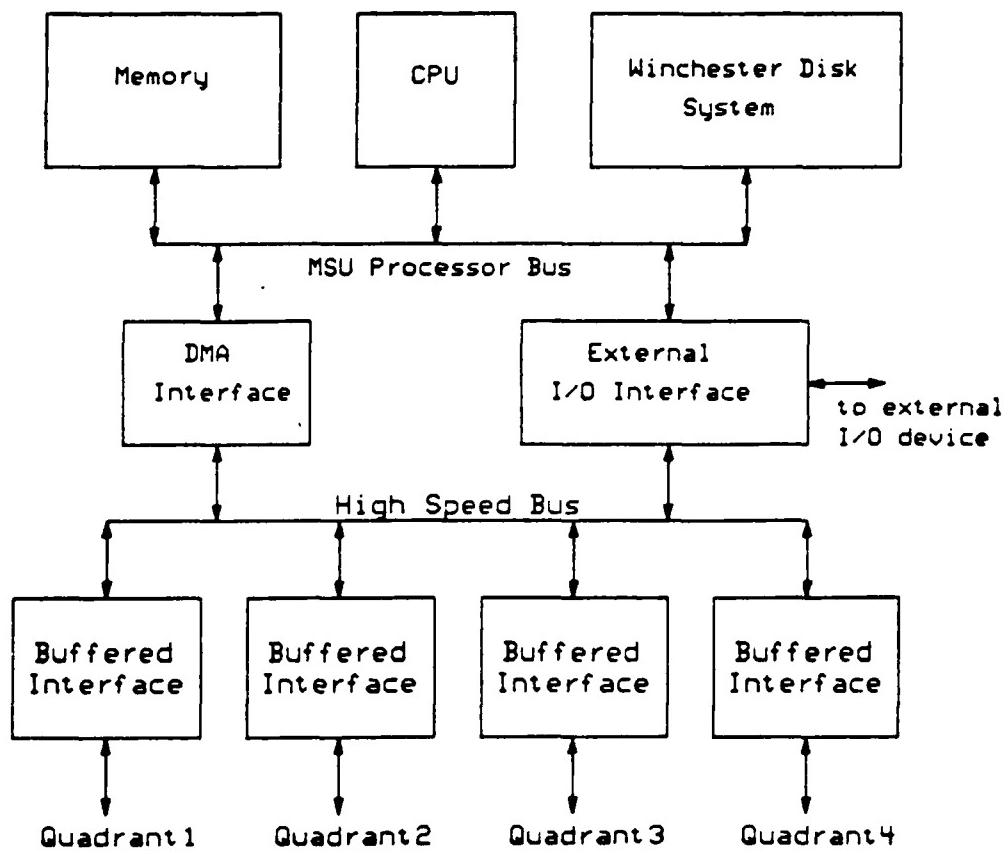


Figure 1.9.4. MSU structure.

switching. Because the most appropriate communication mode is application-dependent, designing the network to operate in both modes is desirable. A similar "hybrid" circuit/packet switched network approach was suggested in [PrK80].

To simplify the design of the network protocol and hardware, two bit-serial lines are used for each interconnection link between interchange box inputs and outputs. This is to avoid the logic that would be necessary to multiplex the lines between the functions of setting the network (interpreting the data as routing tags) and using the network to transfer data. The internal architecture of an LSI chip implementing an interchange box of the "hybrid" Extra Stage Cube network will now be described.

A 2-by-2 interchange box is organized as shown in Figure 1.9.5. The protocol governing packet-switched transfers through an interchange box in stage i is stated as follows: if a box in the previous stage (stage $i-1$) is connected to this box by the upper (lower) Data line and has data to transfer to this box, the box in the previous stage asserts a request on the upper (lower) Handshake line. If the box in stage i has an empty upper (lower) Input Queue, it responds with a grant signal on the upper (lower) Handshake line. The data packet arrives on the Upper Input (UI) or Lower Input (LI) line and is shifted into an empty Input Queue. Simultaneously, a control packet associated with the data packet is shifted into a corresponding control register. The Interchange Box Control examines the Routing (R) and Broadcast (B) bits (part of the control packet) that control the i th stage and determine to which output(s) the data is to be routed. Handshake line(s) requesting the desired output(s) are asserted. When a granting signal is received from the next stage (stage $i+1$), the 2-by-2 crossbar (Figure 1.7.4) is set with C_{1-4} and the data and control packets are shifted out of the Input Queue and into the next stage. The Interchange Box Control is a finite state machine that arbitrates requests and handles conflicts.

In circuit-switching mode, the PEs first generate control packets which propagate through the network establishing connections. The inter-stage request/grant protocol controlling the movement of the control packets is exactly as described above except that there are no accompanying data packets. When the complete path has been established, there is a virtual circuit formed between the input and output ports of the network and data packet

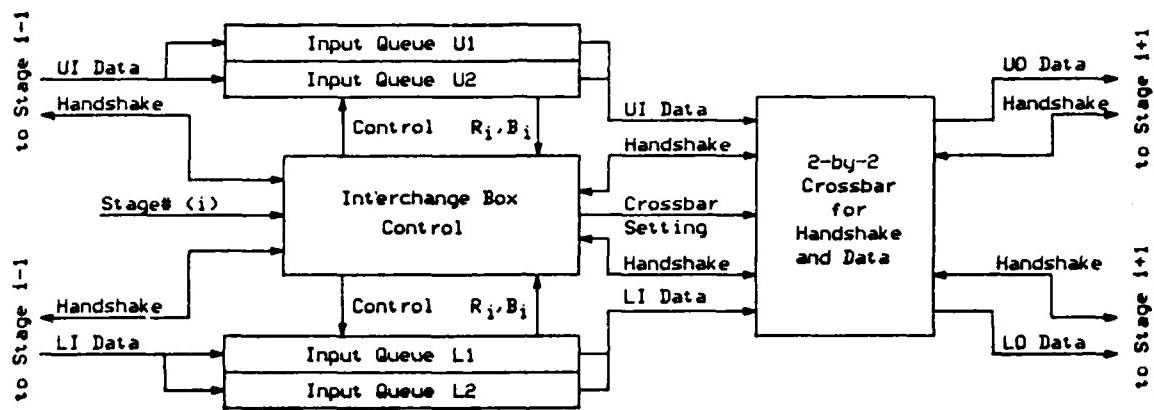


Figure 1.9.5. 2-by-2 network interchange box. The input Handshake lines are comprised of a request signal from and a grant signal to stage $i-1$. The output Handshake lines consist of a request line to and a grant line from stage $i+1$.

transfers may begin. The Data Input Queues are bypassed in circuit-switching mode.

Two possibilities to implement a 4-by-4 subnetwork chip exist. Either four of the 2-by-2 interchange boxes could be interconnected on a chip, or a chip with eight instead of four Data Input Queues and with a 4-by-4 instead of a 2-by-2 crossbar switch could be designed. Studies indicate that the performance of a design with a 4-by-4 crossbar will have higher performance [AdS84c, MaC82]; however, the LSI design will be more complex. For both designs, each network input/output link consists of four signals (data, control, request, grant). Other chip connections required are power, clock input, reset, and stage number input. Another input on the 4-by-4 design would be used to bypass one of the two internal stages. This would allow the chip to be used in forming larger subnetwork modules that have a number of inputs/outputs that is not an even power of two. The 2-by-2 design would require a 28-pin Dual-Inline-Package (DIP) if integrated alone; the 4-by-4 subnetwork module would require a 48-pin DIP.

Given that a 4-by-4 subnetwork module can be integrated on a 48-pin chip, a 32-by-32 subnetwork module can be constructed by interconnecting 24 chips on a board (bypassing an internal stage on eight of the chips). Using the interconnection scheme proposed for the Ultracomputer [GoG83], network stages 0 through 4 of the 1024-PE PASM network would be implemented using 32 32-by-32 subnetwork module boards (chips implementing network stage 4 would have their other stage bypassed. An additional 32 subnetwork module boards would implement stages 5 through 9 (chips implementing network stage 9 would have their other stage bypassed). This is shown in Figure 1.9.6. Thus, all of the extra stage circuitry (4-by-4 chips and extra stage bypass multiplexers) can be physically situated on the horizontal backplane busses. (The extra stage circuitry is not shown in Figure 1.9.3.) The 1024 output links of the extra stage are connected to the 32 32-by-32 subnetwork modules that implement stages 0 through 4 of the network. The 2048 output links of the 32 32-by-32 subnetwork modules that implement stages 5 through 9 of the network are routed back to the PEs. Twice as many links between the output of the network and the PEs are required for redundancy. Still, since each cable "link" consists of only five wires (data, control, request, grant, and ground), this cable

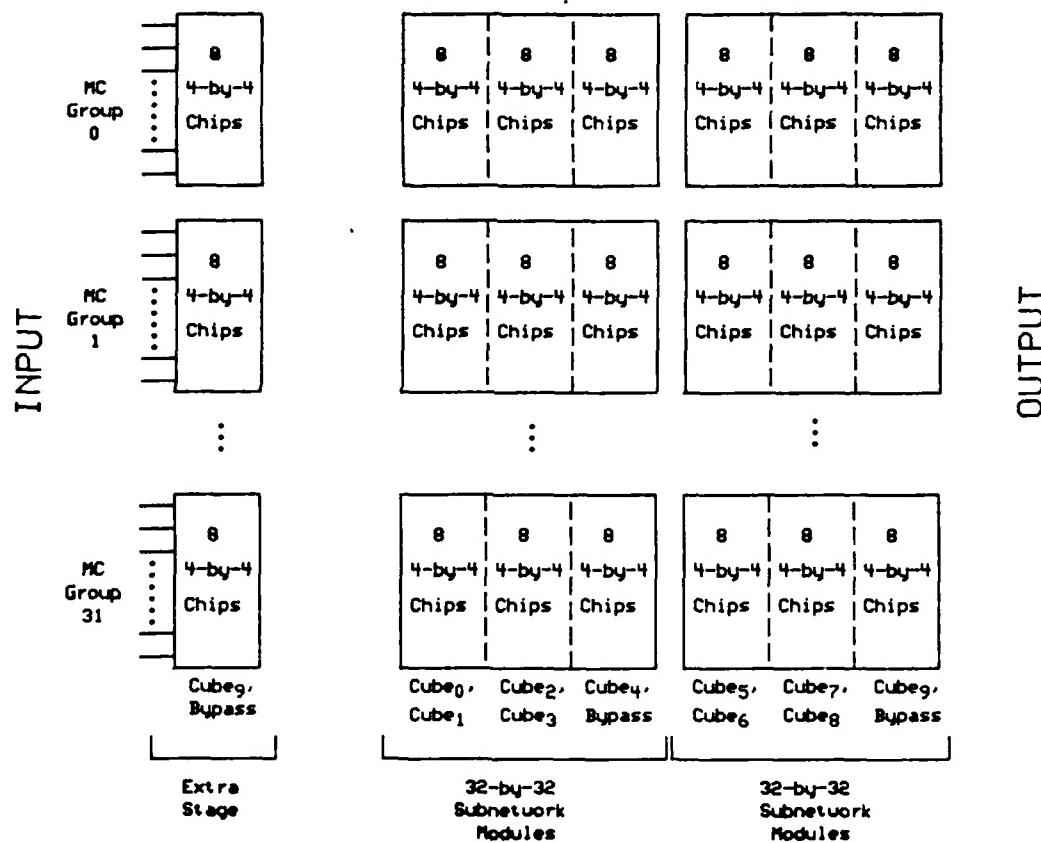


Figure 1.9.6. Extra Stage Cube network layout for the 1024-PE PASM system.

bulk will be manageable. Redundant independent links were not deemed to be necessary between the PEs and the input to the extra stage because the horizontal backplane bus connections are much more reliable than the discrete connecting cables.

Due to the serial approach, speed of the network will be limited. For packet-switching, if a 20 MHz clock speed and a queue length of 64 bits for the network chips are assumed, a message can be strobed into a chip in approximately $6.4 \mu s$. To reach the destination, a message has to pass through $\log_2 N + 1 (=11)$ stages; travel time through the network will therefore be approximately $71 \mu s$. Since messages are queued in every stage, a pipeline effect will result in steady state; assuming no conflicts, packets of a long message will arrive in $6.4\mu s$ intervals. Circuit-switched performance is better since the messages encounter only the propagation delays of the internal 2-by-2 crossbar switches. However, the circuit must be established before it is used. This requires approximately $35 \mu s$ if there are no conflicts: about half of the time required for an end-to-end data packet movement because of the shorter control packets.

Without significantly increasing the cost of the network or complicating the design, the network can be made bi-directional during use in circuit-switched mode. This would result in only a total of four extra return data lines entering and leaving the 2-by-2 chip. If circuit-switched combining operations are allowed as well, the Interchange Box Control circuit needs expanded capability. Reverse packet-switched mode is only useful if the network is to perform combining operations. This essentially doubles the number of pins necessary to implement the 2-by-2 chip.

The three-board set implementing an MC in a 1024-PE PASM system consists of a CPU/memory board; a board for the FBU, its memory, mask decoder, and mask registers; and an I/O board. The CPU/memory board would be similar to the PE board except that the area for the floating point co-processor, Network Interface, and MC Interface would be used for additional memory and a bus interface to the other MC boards. The two other MC boards are scaled-up versions of those used in the prototype.

The floor plan of the proposed system is illustrated in Figure 1.9.7. Inasmuch as the MC68020/MC68881 combination rivals the power of super-

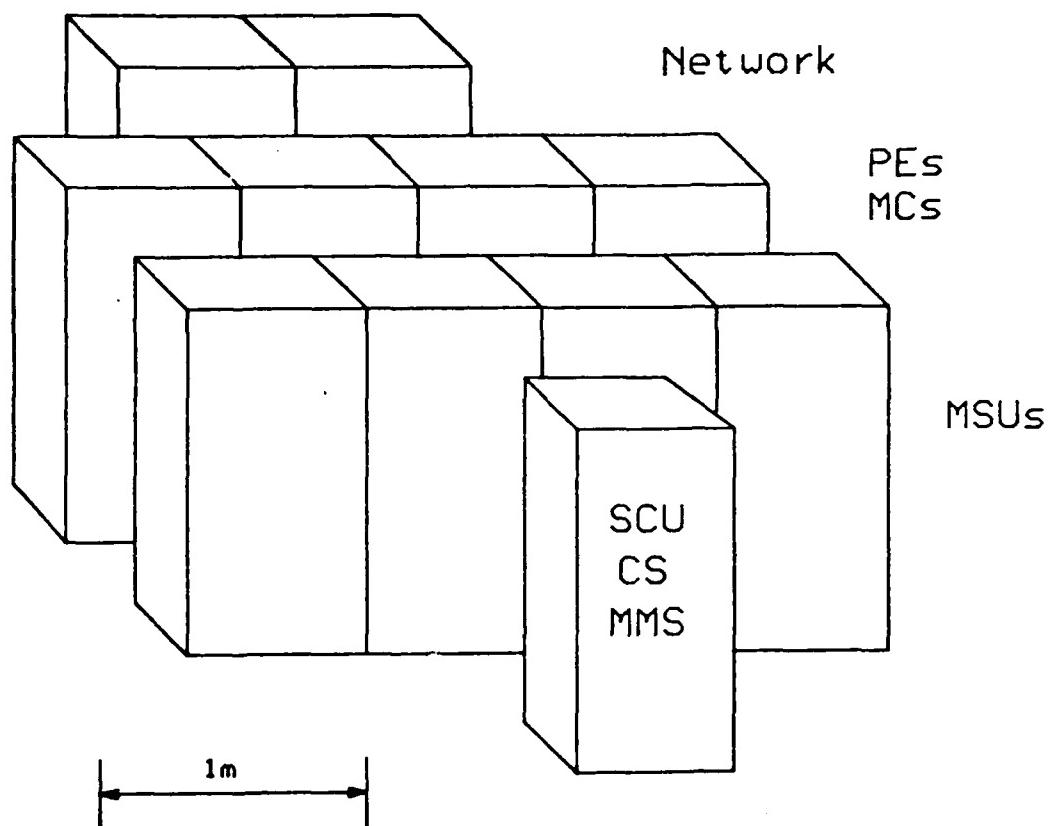


Figure 1.9.6. 1024-PE PASM system floor plan.

mini computers such as a Digital Equipment Corporation VAX, the SCU may be a stand-alone microprocessor system modeled after a PE. Of course, it would require some type of external bus to allow it to be interfaced with secondary storage devices, Ethernet Interfaces, and the like.

It is estimated that the equipment cost for a 1024-PE PASM system would be \$2.7 million; \$1.5 million for the MCs and PEs, \$0.5 million for the hybrid (non-combining) interconnection network (including fabrication cost of the custom LSI chips), \$0.5 million for the MSUs, and \$0.2 million for the System Control Unit and Memory Management System processors. Development time including custom LSI chip design is estimated to be approximately ten man-years.

1.9.4 Summary

A design for a 1024-PE PASM system consisting of a 1024-microprocessor computational engine, multiple secondary storage units, and distributed memory management and control processors has been proposed. Use of off-the-shelf CPUs, memories, and other components in the design of the 30-processor PASM prototype was found to provide the necessary functionality and performance of that system. Their proposed use in the 1024-PE PASM system maximizes the performance/cost ratio because the development cost of these commercial LSI and VLSI components has been amortized over many more units than would be produced for a custom PASM design. Only where space savings are critical and/or the specialized nature of the component makes the functions unavailable commercially have custom LSI approaches been incorporated in the design. Each proposed custom LSI chip design has been examined and determined to be within the capabilities of existing in-house "student" design talent and Computer Aided Design tools.

The PASM architecture, like MPP, achieves its high performance from a sheer multiplicity of computing resources. It does so without the reliance on ultra-sophisticated technologies and hand-crafted "tuned" interconnections found in the vector supercomputers such as the Cray-1 and Cyber-205. Being reconfigurable and capable of both SIMD and MiMD operation, it can be used effectively for a larger class of applications than can an SIMD-only machine

such as MPP. For vector processing, it has more potential computational power than do existing commercial vector/pipeline computers. The capabilities of PASM as a multiprocessor are also significant due to its use of sophisticated processing elements. PASM also has a significant amount of fault tolerance inherent in its architecture: multiple independent SIMD and MIMD machines may be allocated, faulty PEs in a group may be ignored by not scheduling MIMD processes onto them, redundant connections are present in the network, and so on. With this design, it appears that an SIMD/MIMD machine with performance exceeding 1000 MOPS can be readily constructed using current device and interconnection technologies.

CHAPTER 10

SUMMARY

Part I has surveyed existing and proposed parallel processing systems and focused on a particular parallel machine, PASM. A brief history of the PASM development and a detailed description of the design studies performed for PASM was given. These studies helped to influence the design and implementation of the PASM prototype. As a result of the lessons learned from the prototype development, a design proposal for a "full" PASM system with 1024 PEs was made.

PART II

PARALLEL LANGUAGES AND OPERATING SYSTEMS

CHAPTER 1

INTRODUCTION

Because PASM is a research machine and the first of its type (SIMD/MIMD with a PE-to-PE organization), there are many issues in the language and operating system areas to be studied. This part of the thesis discusses the work performed in these areas. Chapter 2 surveys some of the languages that have been proposed or developed for parallel processing systems. An assembler and other support software that have been implemented for use in the PASM prototype are described in Chapter 3. A superset of the C programming language [KeR78] containing constructs for both SIMD and MIMD parallelism is introduced in Chapter 4. Chapter 5 analyzes the characteristics of an operating system for PASM and discusses the hardware and software needed to support them. Part II is summarized in Chapter 6. Some of the material in Part II has been previously published by the author [KuS83, KuS85] or appears in internal reports and manuals [Kue84a, Kue84b] prepared by the author.

CHAPTER 2

SUMMARY OF PASM LANGUAGE AND OPERATING SYSTEM APPROACHES

2.2.1 The Case for Parallel Languages

Parallel languages that have been proposed to date have assumed either a fixed number of processors operating in SIMD mode (e.g., Glypnir [Law75], Actus [Per79], Lucas [Fer82], Vector C [Li84, Li85]) or a generalized multiprocessor (MIMD) model (e.g., ConCurrent C [Nae84], Occam [Inm83]). Designing a language for the SIMD/MIMD PASM model in which programs may use either or both modes of parallelism and a varying number of processors poses a special challenge.

Some language designers have chosen to adopt existing serial languages for parallel machines, relying on extensive analysis during compilation to extract the parallelism (e.g., KAP and VAST analyses of FORTRAN [Arn82]). This approach reduces the "up-front" costs of moving a program to a parallel machine since the program needs only to be re-compiled. However, there is a long-run cost in efficiency since looping and data dependencies implicit in serial languages often obscure the underlying parallelism. Even with better compiler technology, extracting parallelism from existing serial languages will result in less than optimum processor utilization. This is because there is insufficient information to make "safe" assumptions about whether parallelism will lead to read-write or write-write data access conflicts [DiK85].

Had the parallel computer been "discovered" first, we might well be programming in parallel languages exclusively today. The realities of serial hardware have "contorted" the serial language (and perhaps the minds of serial programmers as well). For example, a programmer may spend time trying to

determine how to scan an array of data so that the processing of item i does not cause side effects in the later processing of item $i+1$. This is not because the data *needs* to be processed sequentially, but because the language forces him to do so. Parallel languages allow the programmer to concentrate on algorithm iterations (e.g., iterations to converge on a solution) rather than the processing iterations that are imposed because there are not enough processors to do the job in parallel. The goal of a parallel language and compiler is to free the programmer from thinking about processing iterations. With such a language, even a program to be run on a "parallel" computer having only "one" processor could be coded more quickly in the parallel language since the compiler would add the necessary amount of processing iteration for the single CPU.

By now, the author's bias against the perpetuation of serial languages for parallel computers should be clear. Then why is a serial language [KeR78] being used in this thesis as a starting point for a parallel language? The problem with serial languages is not that they are inherently bad; indeed, many of their features are essential and useful: i.e., control structures, information hiding, decomposition of a problem through procedures, etc. Rather than "starting over," the highly successful C language has been chosen. This language has found favor with systems and applications programmers alike because it is highly efficiently compiled, has modern control structures, and allows the user to get very close to the machine if desired. Because it is a small language (as compared to ADA [Bar82, Led83], for example), compilers for the full language are comparatively available and manageable. In a university environment, this last factor is extremely important.

2.2.2 Language Characteristics

A language provides the programmer with a model of computation. Microprogramming languages present a model of computation based on the generation and test of high and low states of control and data signals in a computer. Each microprogram instruction sets or tests values of such signals. Assembly languages model each basic computer instruction: here data is organized into multi-bit words which are stored in registers or memory and

manipulated by arithmetic instructions. Assembly language is the lowest-level language typically available to a programmer on most computers.

A somewhat higher-level language such as C provides a view of named areas of memory (variables) which are manipulated by arithmetic instructions. At this level, the format of memory is often abstract: it may be thought of as an array of words, a tree of nodes, or as a pointer to a string of characters. Control structures for changing context and looping are provided. Within the C language itself, only type declarations, arithmetic, flow of control operations, and a function call/return mechanism exist. There is no I/O, process control, inter-process communication, or any "built-in" functions of any kind in C. If these functions are needed, they are extracted from operating-system-dependent libraries at program link-load time. This feature of C reduces the complexity of the compiler and makes it more portable; however, the libraries are distinctly non-portable. Use of a particular operating system call from a C program sometimes causes difficulty when the program is ported to another operating system: either the system call may not exist or it may exist in an incompatible form.

An even higher-level language such as ADA includes a variety of built-in functions for I/O, process creation and control, and inter-process communication. Languages at this level are an attempt to free application programmers from operating-system dependencies. Unfortunately, compilers for such high-level languages are operating-system dependent and are typically large, complicated, and slow. Also, high-level languages are often inappropriate for systems programming applications because there is no way to get "close" to the machine.

Languages for parallel computers exist at all of these levels: a parallel language is simply defined as one that allows the programmer to explicitly control more than one computation unit at a time. Conventional microprogramming languages might be considered "parallel" in that they often specify control of several different circuits in a computer simultaneously. An example of a parallel assembly language is given in the next chapter. That language provides the user with a model of an SIMD machine consisting of a scalar unit (the control unit) and a vector unit (some number of PEs). Other parallel assembly languages may provide other models of computation, for example, control over

a pipeline or associative processor. In a later chapter, a parallel version of the C language is given. *Parallel-C* presents the user with a model of an SIMD machine with a scalar and vector/array unit as well as the conventional serial model. If appropriate operating system calls for process creation and communication are provided, MIMD mode processing can also be done. Some recent high-level languages like ADA and Occam already have primitive built-in functions suitable for MIMD mode process handling, but they lack other processing models, notably the SIMD model. SIMD extensions for ADA have been proposed [CIS83, CIS85].

Use of high-level languages saves the human cost of having to retarget software when it is moved. This must be traded off against the long-run cost in efficiency that results from the longer chain of steps required to "translate" the program into machine instructions. In general, use of a lower-level programming language results in greater code density and speed. For PASM, it was determined that in addition to assembly language, some higher-level language should be supported to encourage the use of the machine. In addition, the ease of writing and maintaining the distributed PASM operating system would be enhanced if it were primarily written in a higher-level language. While the goal of "encouraging users" calls for a very high level language and the goal of "efficient systems programming" calls for a lower-level one, a good compromise is C. No doubt, the appeal of implementing and maintaining a single compiler for use by both system and application programmers also played a role in the decision to support the C language on PASM.

2.2.3 Distributed Operating Systems

It is the role of an operating system to provide a "virtual machine" to each user of the computer. A program gains access to the resources and services of the virtual machine such as processors, memory, I/O channels, or a file system via a set of operating system calls. The operating system ensures that resources are allocated fairly and that users are prohibited from interfering with each other.

It is common for the virtual machine interface to be quite different from the physical machine. A simple example is a virtual memory system of a

conventional serial computer: programs may be written to access any address in the virtual space even though the amount of physical primary memory may be a tiny fraction of the size of the virtual space. Another example is the view of many independent users on a conventional multi-user computer system: each "sees" a virtual processor executing his jobs yet there is only one physical CPU being shared. It is entirely possible for a uniprocessor to present users with a virtual parallel processor, although it would do so rather inefficiently.

The role of an operating system for a parallel/distributed computer is the same as for a convention one: to provide a virtual machine to users. The difference is that the operating system for the parallel computer needs to schedule a larger number of resources and must do so in such a way that parallel programs can be run efficiently. Presenting users with a virtual parallel processor of variable size while using a physical parallel processor of a fixed size is a perfectly reasonable goal.

Virtual memory systems are a reasonably good analogue for the desirable goal of providing a virtual parallel processing system because processors are a resource just as is memory. Extending this concept to parallelism introduces the notion of a *Virtual Processing Element (VPE)*. If a parallel program has a declaration of a vector containing N elements, it is most natural to map the vector onto N *Physical Processing Elements (PPEs)*, one element per PPE so that the elements can be accessed and processed in parallel. However, the amount of parallelism desired may be larger than that physically available. A parallel machine with only $N/2$ PPEs can operate on the N -element vector if each PPE simulates two VPEs. Similarly, a uniprocessor can execute the parallel program by simulating the N VPEs. It is natural therefore, to have the program indicate the *extent* of the parallelism (number of VPEs to be used) and for the compiler and operating system to map the needs onto the existing hardware. A scheme for the mapping in PASM is outlined below.

In a PASM system, the number of PPEs is N . This number of PPEs is unchanging: even in the presence of PPE faults or errors which make parts of PASM inoperative, the number of PPEs is assumed to stay the same. If a single PASM PPE becomes faulty, the PASM operating system is restricted to making partitions that do not involve the faulty PPE. Such partitions are limited in size to being $N/2$ PPEs or smaller. Reconfigurable systems such as

PASM also have the notion of *Logical Processing Elements (LPEs)*. In PASM, the number of LPEs assigned to a program is MN/Q , where M is the number of MCs in the machine partition. The responsibility for choosing the mapping of LPEs onto PPEs is completely that of the PASM operating system. Runtime factors such as load, priorities, or faults may affect how the operating system chooses partitions (maps LPEs into PPEs) and partition sizes (how many LPEs are assigned to a particular program). For example, a program given eight LPEs initially may later be forced to run on four mid-way through its calculations due to a high-priority program needing the other four LPEs. These events are completely dictated by the operating system strategy: a general operating system may allow this type of pre-emption; a real-time operating system might not. Operating system calls will exist to allow the user to request increases or decreases in the number of LPEs assigned to a program but such requests are only for the purposes of optimization and may be performed or ignored by the operating system. In non-reconfigurable systems, LPEs are identical to PPEs.

It is the author's philosophy that the user need not be concerned with the actual number of PPEs or LPEs assigned to a task. This is so that a particular state of a machine, for example, the number of working PPEs, the number of non-faulty memory boards, or the system load that day does not affect the results of a computation. Certainly such factors will affect the speed and efficiency with which a computation proceeds, just as they would on a conventional machine. If fewer than the optimum number of PPEs are available, there is a performance penalty, just as there would be in a time-sharing system when more jobs force each to have a smaller slice of the time and memory available. If this philosophy was not adopted for PASM, it would require users to write and compile programs for a number of different partition sizes and to choose which version to run based on the current "health" of the system. This would be a clear barrier to a parallel system's acceptance by users.

The concept of a VPE attempts to free a user from the vagaries of the real machine hardware. The number of VPEs associated with a program may be less than, the same as, or greater than the number of LPEs assigned to it. Conceptually, each VPE is an independent process that is set up by the PASM operating system. The number of VPEs needed by a user's task is determined

by the language compiler or assembler and communicated to the PASM operating system just before execution. However, just as a good programmer tunes his program to a specific hardware configuration (e.g., doing I/O in chunks equal to the physical blocksize of the file system), a parallel program should be tuned to use the number of VPEs that is expected to be physically present in the hardware to enhance efficiency.

The *Virtual Machine* that a user sees for an SIMD program consists of an SIMD control unit and a set of VPEs. An MIMD virtual machine consists only of a set of homogeneous VPEs. As described earlier, the *Physical Machine* that emulates the virtual machine may have completely different characteristics. No further mention of how a physical machine supports the virtual machine is made until the later section describing compilation techniques for PASM.

An operating system for a parallel/distributed computer may be organized in either a centralized or a distributed fashion. Distributed operating systems have been proposed and used where the centralization of operating system facilities would cause a bottleneck in the machine. Such is the case in a large PASM system: the SCU cannot be made totally responsible for monitoring the task status of 32 MCs, much less 1024 PEs. Nor can it have the sole responsibility of memory management, synchronization of PEs in MIMD mode, or control of the interconnection network. Therefore, PASM will have an operating system that is distributed among all of its components.

The main issues in distributed operating systems are:

- (1) how the tasks are to be shared among the multiple processing units;
- (2) how tables and other management information are to be shared and communicated, or if multiple copies are kept, how the copies are updated; and
- (3) what mechanisms for synchronization are provided at both the operating system and user levels.

Existing parallel/distributed systems take very different approaches to these problems mainly because the underlying hardware often dictates certain configurations. Therefore, distributed operating systems will be reviewed in only the most general sense here.

Existing SIMD machines such as the Illiac IV, BSP, and MPP each have a "host" processor that provides the environment for the compilation and debugging of programs, scheduling, and high-level memory management functions.

PASM also relies on a host machine, although in the prototype, a 68000-family microprocessor performs the scheduling and high-level memory management (together with the MMS processors), while a super-minicomputer provides the user programming environment. Of these SIMD computers, only PASM uses a distributed memory management system: the placement of data in the other systems' memory units is controlled directly by the host computer. Further, only PASM has the capability for MSIMD operation; therefore, the problems of dynamically forming different-sized machine partitions must be addressed. This was the main thrust of work performed by Tuomenoksa [TuS81, TuS82b, TuS83, TuS84b, TuS85].

The operating system control schemes that have been studied for MIMD machines are a great deal more varied. There are basically three organizations: *master-slave*, *separate supervisor*, and *floating supervisor* [HwB84]. In master-slave, one processor is responsible for all supervisor functions; all other processors are slaves treated as schedulable resources. Separate supervisor implies that a copy of the kernel is run by each processor as in a computer network. For floating supervisor control, processors operate on supervisory functions on an as-needed basis and control may migrate from processor to processor. In PASM, elements of each of these control schemes are found. An example of "master-slave" control is the SCU task of partitioning the system: only the SCU performs this supervisory function. In MIMD mode, both MCs and PEs act in "separate supervisor" fashion because each manages its own processes and address space. Finally, an example of "floating supervisor" control may arise when different partitions are formed: an MC is the designator for some partitions while for others it is not. While it is the designator it may have extra responsibilities and privileges. A complete summary and design proposal for the PASM operating system is given in a later chapter.

CHAPTER 3

PARALLEL ASSEMBLER AND SUPPORT SOFTWARE

Appendix A1 contains detailed manuals for the support software developed by the author. PASM support software for program development is summarized in the primer "Introduction to 68000-Family Utilities" (Appendix A1.1). Specific information about the PASM (serial) C compiler is given in the "Cc68 Compiler Calling Conventions Guide" (Appendix A1.3). A set of manual pages for the PASM-related software such as the serial C compiler, serial/parallel assembler, serial/parallel linker-loader, preprocessor, object file disassembler, downloader, etc. is also reproduced in Appendix A1.4. Because the C compiler, linker-loader, and certain other related software is derived from versions distributed with the VAX¹/UNIX² 4.2 Berkeley Software Distribution (BSD), copyright and license restrictions prevent their sources from being reproduced here, even in a substantially modified state. The author will provide copies of the support software to organizations holding appropriate VAX/UNIX 4.2 BSD licenses upon request.

2.3.1 Prototype Assembly Language and Assembler

To be able to perform simulation studies of the PASM system executing applications algorithms, one of the first tasks was to develop a language for expressing the algorithms. For MIMD applications, conventional languages are sufficient because each instruction stream executes on just one sequential processor: in this case, a PE. If cooperating PEs in MIMD mode wish to

¹VAX is a trademark of Digital Equipment Corporation.

²UNIX is a trademark of Bell Laboratories.

communicate, operating systems functions are provided to support the interaction. In contrast, SIMD programs cause a set of PEs to operate, each on a different data stream, and all under the control of a sequencing processor (the control unit). The machine model of a scalar unit coupled to a variable-length vector unit is required for SIMD programs. Therefore, a different language syntax is needed to express the parallelism.

The assembly language presented in this section assumes the PASM prototype as a target architecture. Changes in the type of PE or MC from the 68000-family processor to some other processor would necessitate a change in the language, i.e., different assembly instruction mnemonics and operand formats would be used. Also, if the organization of the PASM MC was changed, the type of code the assembler would have to produce would change. Specifically, the assembler knows about the organization and workings of the FBU and automatically produces the special instructions that the MC CPU executes to control the FBU. If the FBU design changes, a different set of controlling instructions would have to be implemented. In addition, the prototype assembler separates the SIMD PE instructions from the MC CPU ones because these are placed in two different memories at execution time.

The parallel assembler, called *pa68*, is derived from a public-domain serial MC68000 one. Over the course of a number of years, *pa68* has been locally updated to support all members of the 68000 processor family. These currently include the MC68000, MC68008, MC68010, MC68012, and MC68020 CPUs and the MC68881 Floating Point Unit. In addition, the support for SIMD parallelism has been added. A complete programmers manual describing the details of the assembler's use in both serial, SIMD, and MIMD modes is found in Appendix A1.2. The source code of *pa68* is given in Appendix A2.1.

2.3.2 Program Sections

A conventional assembler reads instructions that the programmer has coded and produces *object code* for each instruction. An *object file* contains the object code for all the instructions of the program and is typically an image of what physical memory will look like at the beginning of execution. To aid the computer in initializing its memory just before execution of an object file

and in setting up its memory management hardware (if any), the object file is logically divided into a number of *program sections*. A *header* is a fixed-size area at the beginning of the object file that gives the sizes of the sections. An *object file format* is the particular arrangement of the object file header information and the object code for each program section. The object file format is typically hardware- and operating-system-dependent. *Pa68* uses the VAX/UNIX 4.2 BSD object file format in serial mode. This allows VAX/UNIX utility programs such as the linker-loader *ld*, library archiver *ar*, randomizer *ranlib*, and object file pretty-printers *od*, *nm*, and *size* to be used for PASM with only slight modification. A non-standard object file format is used by *pa68* for parallel programs.

The VAX/UNIX object file format consists of a 32-byte header which contains a *magic number* which describes the format of the object file and information for the memory management hardware, the sizes of the *text* (program code), *data* (initialized data), and *bss* (uninitialized data) program sections, the size of the symbol table, the address of the program entry point relative to the beginning of the text section, and the sizes of the text and data relocation command segments. Following the header, the text, data, symbol table, text relocation commands, data relocation commands, and *string table* appear, in order. Only the size of bss is important: space for uninitialized data is allocated just before execution in the computer memory rather than in the object file. The symbol table is made up of fixed-sized entries and is used by a linker-loader to resolve external references and perform relocation. It may also be used by symbolic debuggers. The string table contains the actual (variable-length) symbol names. The text and data relocation commands are present for object files that contain external references. Once object files are linked and loaded, relocation command sections are empty.

The programmer specifies that a block of code or data is to be placed into the text, data, or bss section by issuing an appropriate assembler directive in the assembly language program. The assembler keeps track of the content and size of each segment and creates an object file of the proper format. A linker-loader takes as input one or more object files and rearranges the segments so that all of the text is catenated together, all of the data appears together, and so on. It then writes a new header containing the sizes of the combined

segments and produces a new "composite" object file.

When an object file is to be executed, an operating system process reads it and makes parts of it *resident* in primary memory (the word "loaded" will not be used to avoid confusion with the linking-loading process). Specifically, the operating system reads the header, allocates memory for the text, data, and bss sections, allocates a stack, makes the text and data resident, initializes the bss space to zeroes, and sets the program counter to the entry point location. Memory management hardware is also set, if present, to make the text section execute-only and the rest of the sections read/write. On some hardware, the sections may be placed in different memories. The header, symbol table, and string tables are typically not made resident. PASM performs these actions when an object file is given to a single processor (serial mode) or to a set of processors (MIMD).

SIMD object files are treated similarly, but have more sections and are made resident by multiple processors. PASM SIMD programs specify six different program sections in assembler directives: c_text, c_data, c_bss, p_text, p_data, and p_bss. The "c_" sections are associated with the MC ("c" designates control unit) while the "p_" sections are associated with the PEs. The c_text section is further subdivided into two parts: instruction mnemonics that begin with "c_" are considered MC CPU instructions; mnemonics beginning with "p_" are used to specify instructions to be broadcast to PEs. When such an SIMD program file is assembled, three distinct parts of the object file are formed: one for the MC CPU, one for the FBU, and one for the PEs. Each part has its own header, text, data, symbols, etc. The first part is given to each MC CPU in a PASM partition and consists of the "c_" instructions of the c_text section, the c_data section, and c_bss. These sections contain MC CPU program code, initialized MC data, and uninitialized MC data, respectively. The second part of the SIMD object file is given to each FBU in a PASM partition and consists of the "p_" instructions of the c_text section. No data or bss areas are needed by the FBU. Finally, the last part of the SIMD object file is given to each of the PEs participating in the SIMD machine and consists of the p_text, p_data, and p_bss sections. These sections contain PE program code, initialized PE data, and uninitialized PE data, respectively. The p_text section is used for MIMD code associated with SIMD programs that switch modes.

The difference between SIMD and MIMD mode in PASM is completely defined by where the instructions for the PEs come from. In SIMD mode, instructions are placed by the FBU in a PE instruction queue where they are later broadcast to the PEs. In MIMD mode, instructions are fetched by the PEs from their own memories. For the prototype, PEs fetching instructions from the SIMD Instruction Space are in SIMD mode; otherwise, they are in MIMD mode. Thus if the PEs are currently in SIMD mode and the MC places a "branch to subroutine X" instruction in the queue (where X is some address not in the SIMD Instruction Space), PEs will branch to X and execute instructions out of their own memories (MIMD mode) until a "return" instruction is encountered. The "return" sets the PE program counters back to some address in the SIMD Instruction Space which puts the PEs back into SIMD mode. Alternatively, the MC could send an "unconditional branch to X" to the PEs. It would then be the responsibility of the internal PE monitor for MIMD mode to generate an "unconditional branch to S" (where S is an address in the SIMD Instruction Space) when their work is completed.

The upshot of the above is that SIMD programs may actually be assembled so that certain functions and constructs are actually executed in MIMD mode. Some 68000-family instructions by nature cause an SIMD to MIMD mode switch. An example is "trap" which forces the 68000 into exception processing for the purpose of calling an operating system function. Thus the p_text section is not necessarily empty for SIMD programs.

2.3.3 Prototype Parallel Assembler Criticisms

An unfortunate artifact of the separate MC CPU and FBU memories in the PASM prototype is that the c_text section is split: MC CPU instructions are in one part of the object file, PE instructions to be broadcast are in another. Ideally, the complete c_text section of an SIMD program would be stored in one memory so that the conceptual model of an SIMD machine (scalar unit, vector unit) would be mimicked by the object file format (MC part, PE part).

Another difficulty is that the PASM assembler automatically generates the MC CPU instructions that control the FBU. When the assembler identifies a

block of PE instructions, it determines the block length and generates an MC CPU instruction to write the starting address and word count (length) of the block to the FBU. Giving the assembler knowledge of the FBU hardware so that it could generate such an instruction is both non-portable and non-intuitive in that conventional assemblers need only produce the instructions that the user specifies. On the other hand, forcing users to explicitly write the FBU assembly language commands was deemed to be at least as onerous because it would make all such programs incorrect if the FBU hardware was ever changed.

One possible approach to the FBU instruction generation problem would be to use a preprocessor before the assembler to identify blocks of PE instructions and to insert the FBU instructions where needed. This would "purify" the assembler such that it would no longer need knowledge of the FBU organization. Unfortunately, this approach had to be abandoned because the preprocessor would need to know the length of the block (which is not ordinarily determined until assembly time). Since the "count" register of the FBU is limited to blocks of 255 words, multiple FBU instructions need to be generated for long sequences of PE instructions.

The software problems resulting from the separate SIMD program memories indicate that the prototype FBU organization should be re-thought. From a software point of view, the best solutions are the combined MC CPU/FBU organization and the master MC CPU - slave FBU organization. Each keeps the SIMD instructions together in a single memory and centralizes the control.

2.3.4 Support Software Conversion Experiences

The task of converting a serial utility to an SIMD one typically involved a major rewriting of the code relating to the object file format. The SIMD object file format is non-standard: it consists of four header sections followed by the three major subdivisions for the MC CPU, FBU, and PEs. All headers are of the standard 32-byte length. The first header is special: it contains a magic number that identifies the object file as being in the SIMD format and also indicates the number of VPEs needed for execution. The remaining three

headers are in VAX/UNIX standard format, one for the MC CPU, one for the FBU, and one for the PEs. Taken pairwise, a standard header and its corresponding subdivision are equivalent to what a serial utility would produce or interpret.

It was rather simple to extend those utility programs that processed object files in a sequential way; for example, the object file disassembler and the pretty-printer. This is because the transformation can be performed on each subdivision individually, i.e., first disassemble the MC CPU instructions, then the FBU instructions, then the PE ones.

On the other hand, considerable difficulty was involved in modifying utilities that processed object files by keeping track of text, data, symbol, and relocation areas simultaneously; for example, the assembler and linker-loader. The problem was further exacerbated by coding techniques in those utilities that wrote at several places in the object file simultaneously and that assumed that nothing would ever be written beyond the "end" of a single object file. Also, because the assembler and linker-loader need to process all of the parts of an SIMD object file simultaneously, (e.g., to retrieve PE symbol values that are used in MC code), the internal data structures had to be expanded to accommodate three sets of program section sizes from three different headers, three different location counters, three different sets of file pointers, and so on.

An early PASM parallel assembler and linker-loader implementation used a system of two separate object files rather than a logical subdivision of a single object file. One of the object files would be made resident in the MCs and the other into the PEs. The text section of the MC file was to have been split into an MC CPU part and an FBU part at the time the file was made resident. Unfortunately, this scheme required that the two object files always be handled in pairs and be named in a non-standard way to show their relationship. Because handling files in pairs was completely foreign to related VAX/UNIX support software, it was found that re-writing the linker-loader, library archiver, and other support software was being made much more difficult than would be necessary if a single-file scheme was used. In addition, it was foreseen that the PASM operating system would have to be taught about the special naming conventions of object files. For these reasons, the split-file approach was abandoned.

Additional background and motivation for the particular design chosen for other support programs are given in the manual entries of Appendix A1.4. Experiences with converting the C compiler for SIMD mode operation are described in the next chapter.

CHAPTER 4

PARALLEL-C

2.4.1 Introduction

The success of PASM will depend not only on the presence of reliable and efficient hardware but on the ease of use and adaptability of its languages and operating system. Since much of the operating system for PASM will be developed in the C programming language [KeR78], it seems logical to explore extensions to that language for parallelism. The goal of this chapter is to define a useful and implementable set of extending features that support PASM's modes of parallelism.

A superset of the C programming language that is applicable to the SIMD/MIMD mode processing environment of PASM is described. The language extensions for SIMD mode include the definition of parallel variables, functions, and expressions, an indexing scheme for accessing parallel variables, and extended control structure semantics. Extensions for MIMD mode are realized by broadening the semantics of existing C language constructs, adding a few new keywords, defining a preprocessor to convert the extented language to standard C, and providing additional run-time support in the form of operating system calls. Extensions to the libraries of standard I/O and operating systems functions for use in parallel mode are also discussed.

Some of parallel-C's goals and philosophies are outlined in Section 2.4.2. In Section 2.4.3, new constructs are described for SIMD mode. Section 2.4.4 discussed support for MIMD mode. Some compilation issues are addressed in Section 2.4.5. A number of programming examples are given in Section 2.4.6. Section 2.4.7 contains a formal syntax summary for the language.

2.4.2 Goals and Philosophies

In keeping with the tradition of C, the language extensions proposed attempt to minimize the number of new keywords, constructs, and idioms. Extensions that can reasonably be implemented as a call to a system- or user-supplied function are preferred over new operators that would make the compiler more complex. The C programming language has found favor with systems and applications programmers alike because it is highly efficiently compiled, has modern control structures, and allows the user to get very close to the machine hardware if desired. Because C does not perform run-time bounds or type checking, the language is rather efficient at run time as well. The *laissez-faire* tradition of C is also continued; execution continues unless a program error is non-recoverable: i.e., accesses to non-existent or protected memory, communication with PEs not assigned to the user, corruption of the run-time stack, etc. C's other distinguishing feature, the lack of I/O statements and "built-in" functions in the language, is also continued.

In serial languages, the variable declarations indicate how much memory is to be used by the program. It is natural therefore, to have a parallel program indicate the *extent* of the parallelism (number of VPEs to be used) in the declarations of *parallel* variables and for the compiler or operating system to map the needs onto the existing hardware. Techniques for performing this mapping have been described for other languages (e.g., Vector-C [Li84], Actus [Per79, PeC85]). Because these languages have only vector (one-dimensional) parallel variables, the number of VPEs is given explicitly in the declaration.

The mapping of VPEs to PPEs is done by the compiler for the particular target machines considered so far for these vector languages (Vector-C targets the Cyber 205; Actus targets the Cray-1). When the number of VPEs exceeds the number of PPEs, multiple vector instructions are generated by the compiler to operate on the each part of the vector in a sequential manner. For example, if the number of PPEs is 64 and the vector length is 130, three sequential passes over the vector are needed: first to process elements 0-63, next to process elements 64-127, and finally to process elements 128-129. This approach works only because of the assumptions that there is no reconfiguration nor partitioning. The 64-PPE vector processor cannot be divided among two active processes nor can it be reconfigured into a smaller vector processor if part of it

fails. All 64 PPEs must operate correctly; otherwise, there is complete failure. Obviously, these two conditions do not hold for the PASM system; therefore, PASM must use a VPE to LPE mapping enforced by the operating system.

The author also considers the one-dimensional vector to be too limiting for a "general" parallel language because parallelism exists in physical machines in higher dimensions. For example, MPP and Illiac IV are often considered two-dimensional "array" processors because of their processor interconnection pattern. Pyramid processors extend the notion of parallelism to three-dimensional space. Interconnection networks that allow any source PE to communicate with any destination allow the parallelism to have any conceptual dimension. In Parallel-C, any parallel dimension can be conceptualized and PEs are assigned numbers in each conceptual dimension. A *Conceptual Processing Element (CPE)* of dimension C is identified by a C-tuple giving its number in C -dimensional space. A one-dimensional vector of length V resides in CPEs

$[0], [1], [2], \dots, [V-1];$

a two-dimensional array variable with R rows and C columns resides in CPEs

$[0][0], [0][1], \dots, [0][C-1],$

$[1][0], [1][1], \dots, [1][C-1],$

....

$[R-1][0], [R-1][1], \dots, [R-1][C-1];$

a three-dimensional array variable resides in CPEs identified with three-tuples, and so on.

The mapping a Parallel-C compiler uses to determine which VPE performs the actions of the CPEs from each space depends on the underlying target operating system and hardware. For example, in a vector processor, CPEs in two-dimensional and higher spaces are collapsed to the single dimensional VPE space to be processed. On the other hand, an array processor such as Illiac IV would map CPEs in two-dimensional space into the grid of PPEs to take advantage of the underlying interconnection network. For example, a two-dimensional CPE space of extent $[4][4]$ would be mapped to the first four rows and columns of Illiac IV rather than to all of the columns of the first two rows. Therefore, the mapping of CPE to VPE numbers is target-dependent. This CPE concept allows a compiler to exploit multiple parallel units, e.g., the

multiple vector units in the multi-pipe Cray and Cyber vector processors. The desirability of this was suggested by [LiS85].

2.4.3 Extensions for SIMD Mode

The extensions to C are defined for SIMD mode in the following subsections:

- declarations of parallel variables and functions;
- an indexing scheme to access and manipulate parallel variables;
- expressions involving parallel variables; and
- extended control structures using parallel variables as control variables.

Note that there are no "standard" or "built-in" functions in C: libraries of functions written in C and assembly language are provided at load time. Therefore, functions for data alignment and I/O are treated separately in the final subsection.

Declarations of Parallel Variables and Functions

Some simple variable declarations in standard C are given:

```
int a;                      /* single integer */
char line[MAXLINE];          /* array of MAXLINE characters */
struct node{
    char *word;
    struct node *next;
} nodespace[100], *head;      /* 100 nodes and a pointer to a node */
```

Similar to other parallel languages, both scalar (normal) and parallel variables may be defined. These types of variables are introduced with two new keywords: *scalar* (the default) and *parallel*. All of the examples given above are "scalar" variables since this is the default type. The *scalar* keyword may be applied to any of the declarations above to make the type explicit. The following declarations are equivalent to the examples above:

```

scalar int a;
scalar char line[MAXLINE];
struct node{
    char *word;
    struct node *next;
} scalar nodespace[100], *head;

```

When parallel variables are introduced, they are usually defined in their extent. Situations in which definitions can be deferred to run time will be discussed later. The *parallel* keyword is followed by one or more subscripts that define the *shape* of the parallelism. Assuming N and MAXLINE are compile-time constant expressions, some declarations of parallel variables are:

```

parallel [N] int a;
parallel [N] char line[MAXLINE];
struct node{
    char *word;
    struct node *next;
} parallel [N] nodespace[100], *head;

```

which define for each of N 1-dimensional CPEs one integer *a*, an array of MAXLINE chars, an array of 100 nodes, and a node pointer. The shape of the parallelism may also be multi-dimensional as in:

```
parallel [2][2][2] int a;
```

which declares *a* to be an integer in each of 8 CPEs configured as a 2-by-2-by-2 array (3-dimensional space). Multidimensional parallelism often aids the formulation and coding of certain multi-dimensional problems, for example, image processing.

The keyword *parallel* does not declare a new "type" in the sense of "char" or "pointer to struct node" since both serial and parallel variables might have the same type. Nor does the *parallel* keyword introduce a new "storage class" in the sense of "auto," "static," and "extern." since these are typically data sharing or hiding mechanisms. It is best to consider *scalar* and *parallel* as "storage allocation classes."

Consider the three declarations:

```
parallel [100] struct node pnode;
scalar struct node cnode [100];
parallel [N] struct node ppnode [100];
```

The first two declarations allocate exactly 100 variables of type struct node. The 100 *pnodes* are distributed one to a CPE in one-dimensional space while the 100 *cnodes* are allocated in the SIMD control unit. The last declaration allocates 100N nodes in total, 100 in each of N CPEs in one-dimensional space.

Declaring a variable

```
parallel [1]
```

is equivalent to declaring it

```
scalar
```

in that the same amount of space is allocated, but may have a different effect on how and where the variable is actually stored. This will be discussed in Section 2.4.5.

Parallel [0]

is essentially a no-op. The implications of

```
parallel []
```

will be discussed in Section 2.4.5.

Syntactically, the storage allocation classes (e.g., scalar, parallel) act like type names (e.g., char, float, struct node) and storage classes (e.g., extern, static). Therefore, the storage allocation class keyword need not appear first. For example,

```
parallel [N] static struct node *head;
static struct node parallel [N] *head;
struct node parallel [N] static *head;
```

are all equivalent declarations. The first form will be adopted for the remainder of this discussion.

The type of variable a function returns is known as the "type of a function." For example, the square root function returns a double precision floating

point (double) value. In the calling function, the declaration is:

```
double sqrt();
```

and the function itself defined with:

```
double sqrt(x)
double x;{
    double answer;
    /* calculations */
    return(answer);
}
```

Extending this concept to *parallel* functions, suppose that the square root of a parallel variable is to be calculated. Each CPE will calculate the square root of one element of the parallel variable. The function will return a value in each of the CPEs; therefore, its declaration will be:

```
parallel [] double psqrt();
```

and the definition of *psqrt* is:

```
parallel [] double psqrt(x)
parallel [] double x;{
    parallel [] double answer;
    /* calculations */
    return(answer[*]);
}.
```

Note that the function declaration and definition have empty brackets. This is because functions can be defined to work for any extent of parallelism; the actual extent is determined at run time based on the extents of the function's arguments and returned value. The notation *answer[*]* selects "all of the current extent" and will be discussed in the next section.

As expected, the shapes and types of variables passed and returned to functions should match for correct operation of the program. If there are mismatches, they can be detected by a checking program such as an extended *lint* [Joh78].

Initialization is often done in tandem with declaration. There are two types of initialization: simple and aggregate. Aggregate initialization

(initialization of arrays or structures) is allowed only for external and static variables. Simple initialization is always acceptable.

For scalar variables:

```
int x = 3;                      /* simple */
char line[] = "string";          /* aggregate */
int days[12] = {31, 28, 31, 30, 31, 30,
                31, 31, 30, 31, 30, 31}; /* aggregate */
struct node{
    char *name;
    int id_number;
} persons[] = {
    "joe", 21,
    "sally", 42,
    "jane", 104};
```

For parallel variables:

```
parallel [4] int px = {3, 7, 1, 1};      /* simple */
parallel [4] char pline[] = {"zero", "one",
                            /* aggregate */
                            "two", "three"};

parallel [4] int pdays[12] = {31, 28, 31, 30, 31, 30,
                            31, 31, 30, 31, 30, 31,
                            31, 28, 31, 30, 31, 30,
                            31, 31, 30, 31, 30, 31,
                            31, 28, 31, 30, 31, 30,
                            31, 31, 30, 31, 30, 31};
```

```

struct node{
    char *name;
    int id_number;
} parallel [4] ppersons[] = {"joe", 21,      /* aggregate */
                            "sally", 42,
                            "jane", 104
                            "jack", 117,
                            "kate", 25,
                            "mark", 86,
                            "jeff", 49,
                            "joan", 10};

```

For the initialization of variable *px*, four initializers are given, one for each CPE. As is the convention in C, if fewer initializers than the number needed are given, the rest are assumed to be the value "0." For variable *pline*, the compiler counts the number of characters in each initializer and substitutes the maximum number found for the missing dimension. In this case, six is substituted since the string "three" requires six characters to store the letters plus the null ('\0') character. Thus all CPEs will know line to be a space for six characters, even though some strings required less space (and are padded accordingly). Considering the initialization of variable *pdays*, the rightmost subscript varies fastest. Therefore, the first 12 integers are placed in CPE [0], the next 12 in CPE [1], etc. In the structure node declaration of *ppersons*, the compiler counts the number of initializers and divides the total by four. The result is two initializers per CPE which is the number assumed for the empty brackets. The standard conventions for eliding initializers with braces are applicable here.

Selecting Parallel Variables

When operating on parallel variables, references along the parallel dimension(s) can be done at the same time and will be referred to as *selection*. Selection is typically done using special variables called *selectors* (like the selectors in [Fer82]). A selector is boolean bit vector of any shape that is used to define the set of CPEs that will perform a certain operation.

Selectors are introduced with the *selector* keyword and are similar to parallel variable declarations. For example:

```
selector [N] mask1;  
selector [2][4] mask2;
```

define *mask1* to be a selector for an array of N CPEs in one-dimensional space and *mask2* to be a selector for a 2-by-4 array of CPEs (two-dimensional space). Arrays of selectors, pointers to selectors, etc. may also be defined. For some target machines, the *selector* keyword allows the compiler to generate special instructions so that hardware dedicated to enabling and disabling processors may be used. In others, *selector* may be compiled as a special type of parallel variable. This will be discussed in Section 2.4.5. Selectors are initialized like parallel variables but only 0- and 1-valued initializers are recognized. Any non-zero initializer is taken to be 1-valued.

The initialization of selectors can be a tedious process since the extent of the parallelism can be large. Therefore, a simplified notation based on PE Address Masks [Sie77a] is allowed. For N CPEs, a $\log_2 N = n$ -position mask (*PE address mask*) specifies which of the N CPEs are to be selected. Each position of the mask corresponds to a bit position in the logical numbering of the CPEs and consists of a 0, 1, or X (don't care) specification. CPEs whose addresses match the mask (0 matches 0, 1 matches 1, and 0 or 1 match X) are selected. Mask specifications are grouped with braces, expressions indicate repetition factors, and square brackets denote a complete mask specification. For example, $[n-1\{X\}\{0\}]$ enables all even-numbered CPEs, while $[n-i\{0\}i\{X\}]$ enables CPEs [0] to $[2^i - 1]$.

Some examples of selector initialization are:

```
selector [N] odds = [n-1{X}{1}];
selector [4] odds = [{X1}];
```

```
selector [4] evens = {1, 0, 1, 0};
```

```
selector [4] odds = ![{X0}];
selector [4] odds = {0, 1, 0, 1};
```

```
selector [4] all = [*];
selector [4] all = {1, 1, 1, 1};
```

The first example expands the PE address mask into the aggregate initializer $\{0, 1, 0, 1, 0, \dots\}$ and assigns it to the selector. The second example shows how the aggregate initializer can be specified directly (without a mask). Note the special mask, “[*],” that selects all CPEs in that space. Selectors may be complemented, “or”-ed, “and”-ed, “xor”-ed, or differenced to obtain the complement, union, intersection, equivalence, or set difference of the CPEs selected by each individual selector.

In Actus and Vector-C, the “start:(increment)finish” construct (or its equivalent) and the set operators are used to build “index sets.” This is an equivalent mechanism for expressing the extent of parallelism, but the resulting index sets are static for these languages. Selectors are dynamic and if specified using masks, more space-efficient.

No special notations have been devised for initializing multi-dimensional selectors, but the standard rules applying to aggregate initialization and subscripting are easy to apply to these shapes. For example, the following two declarations are equivalent:

```
selector [2][4] mask2D = {{XX}}, [{X0}]];
selector [2][4] mask2D = {1, 1, 1, 1, 0, 1, 0};
```

and cause each mask to be expanded into its corresponding bit vector and used to initialize the selector as if it were a parallel variable of the same dimensions.

When a parallel variable of a certain shape is referenced, the selector(s) used to select the elements must match the shape. For example:

```
parallel [N] int a;          /* declaration */
a[evens]                   /* reference */
```

selects the value of *a* in the even-numbered CPEs. Similarly,

```
parallel [N] int a[10];      /* declaration */
a[odds][0]                  /* reference */
```

selects the value of *a*[0] in the odd-numbered CPEs. "0" is the index of the first element of the array *a* within a CPE.

A more complex example (*N*=4) is:

```
selector [N][N] mask2 = {[n/2{X}n/2{0}], [n/2{X}n/2{0}], [n{0}], [*]};
parallel [N][N] int a[10];    /* declaration */
```

```
a[mask2][0]                /* reference */
```

for which *mask2* is expanded to

{[X0], [X0], [00], [XX]},

and then to

{1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1}.

Therefore, the value of *a*[0] is selected in CPEs [0][0], [0][2], [1][0], [1][2], [2][0], [3][0], [3][1], [3][2], and [3][3]. Also,

a[evens][evens][0]

selects the value of *a*[0] in CPEs [0][0], [0][2], [2][0], and [2][2].

Of course, a PE address mask or the "[" notation can be used to select a parallel variable directly (without an explicit initialization of a selector variable) as in:

```
a[{XX00}]
a[*].
```

The only restriction is that the shapes of the parallelism be known at compile time.

When scalar variables or constants are used to select parallel variables, only one element of the parallel variable is accessed. Parallel variables can also be used to select and index parallel variables. Since parallel variables are not

necessarily 0- and 1- valued, all of the non-zero elements of the parallel variable are treated as 1-valued for the purposes of selection. Note that any variable used for selecting the range of the parallelism is not itself selected. For the declarations:

```
parallel [M][M] int data[50], index;
selector [M] mask;
parallel [M] pvar;
```

the following are valid references:

```
data[mask][pvar][10];
data[mask][pvar][index[*][mask]]
data[index][3].
```

The first reference causes *data* to be selected in the CPEs along the first dimension specified by *mask* and in the second dimension where the local element of *pvar* is non-zero. The item of *data* within a CPE is determined by the final index "10." The second reference selects the same CPEs as the first, but the indexing within a CPE is controlled by the local value of *index*. The third reference uses the two-dimensional parallel variable *index* in the role of a selector: where *index* is non-zero, CPEs are selected. The particular item within a CPE is chosen with the index "3."

The following are invalid references:

```
data[mask][pvar[*]][10];
data[mask][pvar][index];
data[index[*][*]][3];
```

in the first case because *pvar* is itself selected, in the second case because *index* is not selected, and in the third case because *index* is itself selected.

The type casting operator of C can be extended to change the type and shape of parallel variables. For example, if the variable *y* is a scalar integer, the following would be used to convert it to a double:

```
(double) y.
```

If *y* were a parallel integer,

```
(double) y[*]
```

would be used. If the shape needed to be changed as well, for example, *y* declared as:

```
parallel [N/2][2] int y;;
```

the cast would be:

```
(parallel [N] double) y[*][*].
```

The convention is that the reshaping is done assuming a row-major storage order (rightmost subscript varies fastest) just as it is for standard array indexing and storage in C. Thus the *y* associated with CPE [0][0] is now with CPE [0], [0][1] with [1], [1][0] with [2], ..., and [N/2-1][1] with [N-1]. Depending on the underlying hardware, data may actually be moved from VPE to VPE (and thus PPE to PPE) when reshaped.

As discussed earlier, the type of variable a function returns is known as the type of a function. Since functions can return parallel types, the return values from parallel functions are selected just like ordinary parallel variables. For example, to have each CPE calculate and return the square root of *y* and to select all of the returned values, the following would be used:

```
parallel [N] double y, psqrt();      /* declarations */
psqrt(y[*])[*];                      /* function call */
```

To perform the operation in all CPEs, but to select only the even-numbered CPEs' results:

```
psqrt(y[*])[evens];
```

would be used. Finally, to perform the operation in only the even-numbered CPEs and to select only the even-numbered CPEs' results:

```
psqrt(y[evens])[evens];
```

is used. Note that from the point of view of the caller, the last two forms return the same result. However, functions that are passed different ranges of parallelism, i.e., [*] vs. [even], may produce different side effects due to inter-CPE data transfers, using the extent of parallelism in expressions, etc.

All active CPEs of a given dimensionality evaluate their arguments when a function is called. (Some CPEs might be temporarily inactive due to conditional statements as described later.) Associated with each actual parameter in

the parameter list, there is a selector that indicates which elements of the variable are to be considered "active" and used in expressions. A copy of the value of the parameter and its associated selector is placed on the stack of all active CPEs. In the first two cases above, *y*'s associated selector is [*]; therefore, each active CPE stacks the value of *y* followed by the corresponding element of the selector (each CPE stacks a 1). In the third case, *y*'s associated selector is [evens]; therefore, all active CPEs stack *y* but then even-numbered CPEs stack 1s and odd-numbered CPEs stack 0s.

For each formal parameter of a given type declared in the called function, there must be a corresponding actual parameter of the same type in the call. Assume that within the *psqrt()* function, *spy* is the name of the formal parameter. While every CPE active at the time of the call to *psqrt()* "executes" the function, accesses to *spy* are restricted to those CPEs for which *spy* is selected (CPEs having the corresponding selector '1' on the stack). For example, if the actual parameter *y* had selector [evens] as in "psqrt(*y*[evens])," but inside the function, *spy* was used in the expression "*spy*[*]," the value of *spy* would be selected only for the even-numbered CPEs. If inside the function, *spy* was used in an expression like "*spy*[*first4*]," where *first4* selected CPEs numbered [0]-[3], the value of *spy* would be defined only for CPEs [0] and [2]. This is because within the scope of the called function, *spy* was defined only for even CPEs; the *first4* selector further narrowed the selection set within the function. This mechanism allows selection of expressions within a function.

The conventions for returning values are similar. An expression returned by a function may be defined for all CPEs, for example, "return(answer[*]);," but only those results in the CPEs selected by the calling function are used.

Expressions and Assignment

Scalar and parallel variables in expressions and assignments can be combined as long as no parallel shape conflicts occur. Expressions containing parallel variables always result in a parallel value. Scalars are "promoted" to parallel type when used in expressions with parallel variables just as "chars" are promoted to "ints," "floats" to "doubles," etc.

There are four cases of assignment statements:

1. Left side scalar; right side scalar. This is the normal C assignment statement.
2. Left side scalar; right side parallel. The right side of the assignment should be indexed so that one and only one element is selected. Selecting other than one element is *not* an error, but the results are undefined.
3. Left side parallel; right side scalar. The right side is promoted to the parallel type and shape of the left side. Selected elements of the left side are assigned the value of the scalar.
4. Left side parallel; right side parallel. The selected elements of the left side are assigned the corresponding values of the right side expression. The range of the selected items on the right side should equal or overlap that of the left for the results to be defined.

For example, if $N = 4$, $n = 2$:

```
selector [N] evens = [n-1{X}{0}];  
parallel [N] int a, b;
```

```
b[evens] = a[evens];
```

assigns the value of a to variable b in CPEs [0] and [2]. For:

```
parallel [N] int a[10], b[10];
```

```
b[evens][0] = a[evens][0];
```

the value of $a[0]$ is assigned to $b[0]$ in CPEs [0] and [2].

A more complex example ($N=4$) is:

```
selector [N][N] mask2 = {[n/2{X}n/2{0}], [n/2{X}n/2{0}], [n{0}], [*]};  
parallel [N][N] int a[10], b[10];
```

```
b[mask2][0] = a[mask2][0];
```

in which $a[0]$ is assigned to $b[0]$ in CPEs [0][0], [0][2], [1][0], [1][2], [2][0], [3][0], [3][1], [3][2], and [3][3]. Also,

```
b[evens][evens][0] = a[mask2][0];
```

assigns $a[0]$ to $b[0]$ in CPEs [0][0], [0][2], and [2][0]. In CPE [2][2], $b[0]$ was set

to garbage (undefined) since the corresponding element of $a[0]$ was not accessed. In CPEs [1][0], [1][2], [3][0], [3][1], [3][2], and [3][3], values of $a[0]$ were accessed but not written into $b[0]$.

Even though there was a mismatch in the ranges of parallelism selected in the last statement, there is no syntactic or run-time error condition generated. It is often useful to select more items on the right-hand side of an assignment than are selected on the left. Since some elements of b were made undefined by this operation, it may represent a programmer error, but execution will be allowed to continue. Such a policy may seem counter-intuitive, but frequently such mismatches can be used to increase program efficiency through incomplete selection. Programs will function correctly if undefined results are not depended on.

Recall that in an assignment where both the right and left sides are parallel, *corresponding* elements are assigned. Thus

parallel [N] int a, b;

a[2] = b[3];

does *not* move the value of b from CPE [3] to a in CPE [2]. In fact CPE [2]'s a becomes undefined since no element of the right hand side is selected for CPE [2]. Special functions (described later) are used to move data values from CPE to CPE.

Control Structure

C has five different constructs for affecting control flow: *if-then-else*, *switch-case*, *while*, *for*, and *do*. The first two are used to select different execution paths, while the remaining constructs control repetition of statements. The semantics of each of these constructs have been extended for parallel mode.

The construct:

```

if ( <scalar-expr> )
    <statement1>;
else
    <statement2>;

```

selects one of the two statements based on the truth value of the expression. In the corresponding parallel construct:

```

if ( <parallel-expr> )
    <statement1>;
else
    <statement2>;

```

the <parallel-expression> can be thought of as a selector which is “true-valued” for some CPEs and “false-valued” for others. Thus some of the CPEs will perform <statement1>, while others will execute <statement2>. In a strict SIMD environment, <statement1> and <statement2> cannot be executed by the CPEs simultaneously since there is a single instruction stream. Thus, side effects from <statement1> may affect <statement2>. PASM, which can operate in SIMD or MIMD mode, might temporarily switch to MIMD mode so that <statement1> could be executed in parallel with <statement2>. This will be discussed further in Section 2.4.5.

The programmer should be on the lookout for situations in which the *if* statement can be avoided. For example,

```

selector [N] a_larger;
parallel [N] int a, b;

if( a[*] > b[*] )
    a_larger[*] = 1;
else
    a_larger[*] = 0;

```

is better written as:

```
a_larger[*] = a[*] > b[*];
```

The parallel equivalent of the *switch-case* statement for takes the following form:

```

switch( <parallel-expr> ){

    case <scalar-constant1>: <statement1>; break;

    case <scalar-constant2>: <statement2>;

    case <scalar-constant3>: <statement3>; break;

    ...

    default:           <default-statement>;
}

```

Here, each CPE evaluates the *<parallel-expression>*, derives its own result, and executes the statements in the appropriate case. In a strict SIMD environment, the CPEs that are to execute each case are selected (enabled) and the statements for that case executed. The CPEs that will execute the next case are then selected, and so on for each case sequentially. As with the *if* statement, side effects resulting from the execution of earlier cases may affect later cases. The semantics of the *switch* statement in C are that execution of a case begins at the matching case label and continues until the end of the switch unless a *break* statement is encountered. For this reason, CPEs may execute the code for more than one case. In the example above, any CPE executing *<statement2>* also executes *<statement3>*. In the PASM environment, cases may be executed by all PEs simultaneously.

The parallel *while* statement takes the form:

```

while ( <parallel-expr> )
    <statement>;

```

Repetition continues as long as *any* element of the *<parallel-expression>* is true. The statement is only executed by those CPEs selected by the *<parallel-expression>* for the current iteration.

The parallel *do* statement takes the form:

```

do
    <statement>;
    while( <parallel-expr> );

```

Repetition continues as long as *any* element of the <parallel-expression> is true. The statement is performed in every CPE at least once, and continues for as long as required in those CPEs selected by the <parallel-expression>.

The parallel *for* statement has the form:

```

for( <parallel-expr>OPT; <parallel-expr>OPT; <parallel-expr>OPT )
    <statement>;

```

The second <parallel-expression> is evaluated at the beginning of each iteration and selects the CPEs that are to be active (perform the statement) for that iteration. In typical usage, the first <parallel-expression> is an assignment statement that initializes the index variable. The third expression typically re-initializes or increments the index variable. In the parallel environment, CPEs may initialize their index variables to different values and increment them by varying amounts at each iteration. Thus, each CPE may undergo differing numbers of repetitions of the statement, just as for the *while* and *do* statements.

Functions for Data Alignment and I/O

Each of the functions described below would be contained in a library that would be available at link-load time. This list is not meant to be exhaustive since there are many data alignment patterns and I/O protocols for SIMD mode would be useful for some circumstances. Note that some functions are only defined for certain shapes and extents of parallelism. Some of the following functions are *system calls (syscalls)* which are the basic entry points of the underlying operating system. Syscalls execute in MIMD mode; they return a value in each PE that calls them. Syscalls are identified by the type definition "SYSCALL." Other functions not so identified can be written in an operating-system-independent way.

Before presenting a set of functions that implement a general inter-CPE communication scheme, a further discussion of CPE numbers is in order. A

Parallel-C program may contain parallel variables of a number of different shapes and extents although it is quite common that only one parallel shape is used in a program. The shapes and extents of all Parallel-C variables are defined at compile time. The number of VPEs associated with the program is decided by the compiler but is guaranteed to be at least as large as the number of CPEs called for by any single parallel variable. Thus if variable *oneD* has a shape "parallel [32]" and variable *twoD* has a shape "parallel [4][4]," at least 32 VPEs are assigned. To allow for optimizations, the compiler may round the number of VPEs up to a machine-dependent value. For example, in an 8-by-8 mesh-connected computer that does not allow partitioning, the compiler may always make the number of VPEs a multiple of 64.

How CPE numbers are mapped to VPEs is also left up to the compiler. For example, while *oneD[0]* is in CPE [0] and *twoD[0][0]* is in CPE [0][0], these two CPE numbers are quite distinct and are not necessarily both mapped to VPE 0. To allow for optimizations, the CPE numbers may be assigned to the VPEs in different ways depending on the shape of the parallelism. An example of an optimization for an 8-by-8 mesh-connected computer might be to map the variable *twoD* onto the two-dimensional machine structure so that the first index indicates a row number and the second index a column number in the target machine. Therefore, *twoD* would be mapped into VPEs in the first four columns of the first four rows of the machine while *oneD* would be mapped into the VPEs in the first four rows consecutively. The mapping rules a compiler uses will be highly dependent upon the processor interconnection scheme of the target machine.

In general, the programmer need not worry about the mapping of CPEs to VPEs. Hence, if a variable is used such that all of its elements are selected, i.e., "twoD[*][*]," VPEs actually holding an element of the variable will be involved in the computation. Other VPEs in the machine partition will either be disabled or will perform the computation in such a way so as not to interfere with the intended results. Users deal with CPE numbers exclusively.

In the following data alignment function declarations, *dimension* is a variable of type "parallel [] int" that indicates the number of dimensions of a parallel shape. For example, "parallel []" has a dimension of one, "parallel [][]" has a dimension two, etc. *Place* is a variable of type "parallel [] int *" that points to

an array of *dimension* integers that hold the indices of a CPE. A source or destination CPE number can then be specified by the 2-tuple (dimension, place). As an example, suppose that *dimension* is 3 and *place* points to an array containing the integers 1, 2, and 3. This 3-tuple specifies the CPE numbered [1][2][3]. Note that the arrangement and type of the *dimension* and *place* variables in memory is not specified by Parallel-C in any way; it is simply used to communicate a particular CPE number to the operating system. Although users need deal only with CPE numbers by calling library functions, PPE, LPE, and VPE numbers are manipulated within the library functions to perform the I/O and data alignment operations. Obviously, users can bypass any of the top-level library functions and make syscalls to obtain and manipulate the LPEs and VPEs explicitly. Recall that for non-reconfigurable systems, LPEs are identical to PPEs; therefore, some operating systems might not implement syscalls that use LPE numbers.

SYSCALL parallel // int get_ppe()

Returns a PPE number. The number returned to a given PPE will never change.

SYSCALL parallel // int get_lpe()

In PASM and other reconfigurable systems, an LPE number is returned; otherwise, the PPE number is returned. An LPE number may change during the course of execution of a program because the operating system may assign more or fewer PPEs depending on load, priority, or other factors.

SYSCALL scalar int get_M()

Returns the number of MCs executing the current process. The number may change during the course of execution of a program because the operating system may assign more or fewer PPEs depending on load, priority, or other

factors. This call is specific to PASM. Technically, this call is superfluous because M can be derived from the relationship between the LPE and PPE numbers.

SYSCALL parallel [] int get_vpe()

Returns a VPE number. VPEs are mapped to LPEs (or PPEs) by the operating system and each LPE in a partition may simulate one or more VPEs.

```
parallel [] void get_cpe(dimension, place)
parallel [] int dimension;
parallel [] int *place;
```

Determines the CPE number of the caller expressed in the desired *dimension*. The indices are placed in the array pointed to by *place*. This CPE number can then be manipulated to obtain the CPE numbers of neighbors in a relative way. For example, consider two-dimensional CPE numbers of the form “[row][column]” for a parallel variable with shape and extent “parallel [R][C].” To form a CPE number of the “row above” (using modulo R arithmetic) let each CPE subtract one (modulo R) from the first integer in the *place* array. This gives the number of a CPE in the previous row for all two-dimensional CPEs that execute the function.

The *get_cpe()* function calls *get_vpe()* to determine which VPE the caller is currently simulating. This call is compiler-dependent because each compiler has a different scheme for mapping CPE numbers of a given dimension to VPE numbers.

```
parallel [] int cpe_to_vpe(dimension, place)
parallel [] int dimension;
parallel [] int *place;
```

Takes the CPE number expressed in the desired *dimension* and returns the

VPE number in which it appears. The indices are placed in the array pointed to by *place*. This function is typically called as the first step of an inter-CPE transfer function to determine a destination VPE number. The next step would be to perform a syscall to request a transfer between the two VPEs. This call is compiler-dependent because each compiler has a different scheme for mapping CPE numbers of a given dimension to VPE numbers.

```
parallel // void vpe_bcopy(vpe, psrc, pdest, nbytes)
parallel // int vpe;
parallel // char *psrc, *pdest;
scalar unsigned nbytes;
```

This function copies *nbytes* bytes of data from the calling (source) VPE to the destination VPE specified by *vpe*. The set of source VPEs must map one-to-one and onto the set of destination VPEs; otherwise, a run-time error is generated. Source bytes are read beginning at the location specified by *psrc*. Incoming data is written beginning at the location specified by *pdest*.

The operating system local to each PPE determines where (which LPE or PPE) the destination VPE resides and sets the interconnection network accordingly. Because each LPE or PPE may be simulating multiple VPEs, not all inter-VPE transfers necessarily use the interconnection network. As a trivial example, consider a sequential machine simulating *V* VPEs: inter-VPE transfers for this machine consist of memory-to-memory copy operations.

This function will make two syscalls: one to write the data to the destination VPE and one to read incoming data to the VPE. Because of the VPE time-slicing when there are insufficient LPEs, the controlling MC instructions associated with this function force all VPEs to perform the write syscall before any are allowed to perform the read syscall. If this was not done, a destination VPE might attempt to read data that has not been written yet because the source VPE had not yet received a time-slice. Failure to synchronize SIMD reads and writes would also leave open the possibility of network conflicts.

```
parallel [] void cpe_bcopy(dimension, place, psrc, pdest, nbytes)
parallel [] int dimension;
parallel [] int *place;
parallel [] char *psrc, *pdest;
scalar unsigned nbytes;
```

This function copies *nbytes* bytes of data from the calling (source) CPE to the destination CPE specified by *dimension* and *place*. The set of source CPEs must map one-to-one and onto the set of destination CPEs; otherwise, a run-time error is generated. Source bytes are read beginning at the location specified by *psrc*. Incoming data is written beginning at the location specified by *pdest*. This library function would be implemented by a call to *cpe_to_vpe()* to obtain a VPE number followed by a call to *vpe_bcopy()*.

```
parallel [] int cpe_move(dimension, place, data)
parallel [] int dimension;
parallel [] int *place;
parallel [] int data;
```

This function moves the integer *data* from the calling (source) CPE to the destination CPE specified by *dimension* and *place*. The set of source CPEs must map one-to-one and onto the set of destination CPEs; otherwise, a run-time error is generated. The function returns the data transferred in the destination CPEs. This library function would be implemented by a call to *cpe_to_vpe()* to obtain a VPE number followed by a call to "vpe_bcopy(vpe, &data, &data, sizeof(int))." The value of *data* after *vpe_bcopy()* returned would then be returned to the caller.

The following functions are intended to be representative of those that might be used to communicate among a one-dimensional vector of *C* CPEs.

```

parallel // int broadcast_1D (destaddrlist, C, data)
parallel // unsigned char *destaddrlist;
scalar unsigned C;
parallel // int data;

```

The *data* value(s) from the source CPE(s) that is (are) selected is (are) broadcast to the *C* destination CPEs specified in the *destaddrlist*. The *destaddrlist* in each selected source CPE is taken to be an array of C 0- and 1-valued elements. Source CPEs broadcast to all destination CPEs numbered *i* where the element *i* of the array is 1-valued. Unselected source CPEs do not broadcast anything. It is a run-time error to have a destination CPE receive more than one item.

In PASM, the types of broadcasts that can be performed in one step by its multistage cube-type network are restricted. This very general function provides a common interface for all networks.

```

parallel // int move_1D (cpe, data)
parallel // int cpe;
parallel // int data;

```

This function is equivalent to performing “*cpe_move(1, &cpe, data)*.” Only CPEs where *cpe* is selected transfer data.

```

parallel // unsigned bcopy_1D (cpe, psrc, pdest, nbytes)
parallel // int cpe;
parallel // char *psrc, *pdest;
unsigned nbytes;

```

This function is equivalent to performing “*cpe_bcopy(1, &cpe, psrc, pdest, nbytes)*.” Only CPEs where *cpe* is selected transfer data. The function returns the number of bytes actually transferred.

```

parallel // int shift_1D (data, shiftcount)
parallel // int data;
scalar int shiftcount;

```

This function takes an integer *data* item from each participating source CPE and returns their values in the destination CPEs. *Shiftcount* indicates the amount and direction of the movement of data: data is moved from CPE *i* to CPE *i+shiftcount*. For positive (negative) *shiftcounts*, the first (last) *shiftcount* CPEs selected receive 0-valued data.

```

shift_bcopy_1D (psrc, pdest, nbytes, shiftcount)
parallel // char *psrc, *pdest;
scalar unsigned nbytes;
scalar int shiftcount;

```

This function moves a block of *nbytes* bytes starting from location *psrc* in the source CPEs to location *pdest* in the destination CPEs. *Shiftcount* indicates the amount and direction of the movement of data: data is moved from CPE *i* to CPE *i+shiftcount*. For positive (negative) *shiftcounts*, the first (last) *shiftcount* CPEs selected receive 0-valued data.

```

parallel // int rotate_1D (data, shiftcount, C)
parallel // int data;
scalar int shiftcount;
scalar unsigned C;

```

This function takes an integer *data* item in each participating source CPE and returns their values in the destination CPEs. *Shiftcount* indicates the amount and direction of the movement of data: data is moved from CPE *i* to CPE $(i + shiftcount) \text{ modulo } C$.

```

rotate_bcopy_1D (psrc, pdest, nbytes, shiftcount, C)
parallel [] char *psrc, *pdest;
scalar unsigned nbytes, C;
scalar int shiftcount;

```

This function moves a block of *nbytes* bytes starting from location *psrc* in the source CPEs to location *pdest* in the destination CPEs. *Shiftcount* indicates the amount and direction of the movement of data: data is moved from CPE *i* to CPE (*i+shiftcount*) modulo *C*.

```

parallel [] int shuffle_1D (data, C)
parallel [] int data;
scalar unsigned C;

```

This function takes a *data* item from each participating source CPE and returns their values in the destination CPEs. The shuffle interconnection function defines the movement of data from CPE *i* to CPE (*i<<1* | ((*i>>c-1*)&01) where *c* is $\log_2 C$ (a left shift end-around shift is performed on the CPE numbers). *C* must be a power of two.

```

shuffle_bcopy_1D (psrc, pdest, nbytes, C)
parallel [] char *psrc, *pdest;
scalar unsigned nbytes, C;

```

This function moves a block of *nbytes* bytes starting from location *psrc* in the source CPE to location *pdest* in the destination CPE. The shuffle interconnection function defines the movement of data from processor *i* to processor (*i<<1* | ((*i>>c-1*)&01) where *c* is $\log_2 C$ (a left shift is performed on the CPE numbers). *C* must be a power of two.

```

parallel // int exchange_1D (data, C)
parallel // int data;
scalar unsigned C;

```

This function takes an integer *data* item in each participating source CPE and returns the values to the destination CPEs. The exchange interconnection function defines the pairwise exchange of data between even and odd CPEs. *C* must be an even number.

```

exchange_bcopy_1D (psrc, pdest, nbytes, C)
parallel // char *psrc, *pdest;
unsigned nbytes, C;

```

This function moves a block of *nbytes* bytes starting from location *psrc* in the source CPE to location *pdest* in the destination CPE. The exchange interconnection function defines the pairwise exchange of data between even and odd CPEs. *C* must be an even number.

```

parallel // int cube_1D (data, i, C)
parallel // int data;
scalar unsigned i, C;

```

This function moves an integer *data* item from each participating source CPE and returns the values to the destination CPEs. The cube interconnection function defines the movement of data from CPE P to CPE $\text{cube}_i(P)$. Expressing the CPE address, P, as a binary number ($c = \log_2 C$): $\text{cube}_i(P = p_c p_{c-1} \dots p_i \dots p_1 p_0)$ is $p_c p_{c-1} \dots \bar{p}_i \dots p_1 p_0$. *C* must be a power of two and *i* must be in the range $0 \leq i < c$.

```

cube_bcopy_1D (psrc, pdest, nbytes, i, C)
parallel // char *psrc, *pdest;
scalar unsigned nbytes, i, C;

```

This function moves a block of *nbytes* bytes starting from location *psrc* in the source CPE to location *pdest* in the destination CPE. The cube interconnection function defines the movement of data from CPE P to CPE $\text{cube}_i(P)$. Expressing the CPE address, P, as a binary number ($c = \log_2 C$): $\text{cube}_i(P = p_c p_{c-1} \dots p_i \dots p_1 p_0) = p_c p_{c-1} \dots \bar{p}_i \dots p_1 p_0$. C must be a power of two and i must be in the range $0 \leq i < c$.

The following functions are intended to be representative of those used to form one-dimensional selectors.

selector // *first(x)*
selector // *x*;

Finds the first true element in the selector *x* and returns a new selector with only this element true.

selector // *next(x)*
selector // **x*;

Finds the first true element in the selector pointed to by *x* and returns a new selector with only this element true. In addition, the first true element in the selector passed is assigned the value false.

scalar unsigned any(x)
selector // *x*;

Returns the number of non-zero elements in the selector.

```
scalar unsigned all(x)
selector // x;
```

Returns the extent of the selector if all elements of the selector are non-zero; else returns 0.

Finally, libraries may also contain functions intended for higher-dimensional parallelism. Some examples for two-dimensional parallelism might be:

```
parallel [][] north_2D
parallel [][] south_2D
parallel [][] east_2D
parallel [][] west_2D
parallel [][] rowexchange_2D
parallel [][] colexchange_2D
parallel [][] transpose_2D.
```

Although these will not be documented here in detail, all of these can be derived by calls to "get_cpe()" to determine the local CPE number, modifications of the CPE number to determine the relative movement (i.e., north, south, transpose), and finally a call to "cpe_bcopy()" or "cpe_move()" to perform the transfer.

I/O and Mathematical Functions

Processors may call I/O and mathematical functions that accept parallel variables. For example, the "psqrt()" function described in an earlier section accepts a parallel [] double and returns a parallel [] double. In general, library functions for SIMD mode will be the same as for serial mode, but will accept parallel data, return parallel data, and be named by prefixing a "p." I/O and mathematical functions that require special shapes may be defined by the user by shape-casting the parallel functions provided.

2.4.4 Extensions for MIMD Mode

The constructs given in the last section allowed a single program (instruction stream) to access and manipulate data items located in multiple VPEs. MIMD mode implies that each processor will execute its own instructions from its own memory; thus none of the SIMD constructs are necessary in MIMD mode: MIMD programs will compile using the standard C compiler. However, MIMD processes must coordinate by communicating with each other and by synchronizing the order of their execution. Furthermore, processes operating on shared data structures must do so in a mutually exclusive manner to maintain data integrity.

The author proposes that a *meta-language* such as ConCurrent C [Nae84] be developed for use on PASM in MIMD mode. This meta-language contains new constructs and keywords that are meant to express useful operations that are typically performed on multiprocessing systems. The extensions do not affect the C language nor its compiler: a preprocessor is used to convert the meta-language to standard C. The preprocessor is operating-system-dependent because the translation process introduces syscalls into the standard C code to perform some of the functions required. Of course in PASM, MIMD and SIMD functions must co-exist: SIMD programs must be allowed to call MIMD functions and vice-versa. Therefore, the eventual goal is a preprocessor that accepts a set of SIMD/MIMD programs, converts the MIMD portion to standard C, and calls the extended (SIMD) compiler to compile the SIMD and standard C.

It is useful to review the features of ConCurrent C at this point. In ConCurrent C, two new classes of constructs are proposed:

1. variable declarations: event variables and shared variables
2. control flow statements: for process interaction, control, concurrent execution, and event supervision.

Real-time *events* are triggered by *timers* elapsing or by an *exception*. *Event variables* are introduced with the keywords *timer* and *exception* and are *activated* (enabled) with a call to a built-in function "activate." Note that the meta-language has built-in or reserved function names; however, this does not imply that the underlying C language does. *Internal exceptions* such as underflow, divide by 0, kill, etc. are predefined; the program may specify

others. *External exceptions* are defined for an arbitrary set of processes and can initiate signaling of an event to other processes. *When* is a keyword that evaluates a condition and suspends the process until a specific event is satisfied. When the condition is satisfied, the statements in the associated block are executed. *When-else* is similar, but does not suspend if the event is not satisfied: statements in the "else" block are done instead. *Whenever* forks off a separate *event observer* and defines an *event handler* that is executed when the event occurs. *When* is useful for synchronization, mutual exclusion, and the implementation of critical sections. C's external and external-static variables are considered shared variables. Concurrent execution of processes is provided by the *cobegin* keyword. Calls to the *process()* built-in function start up new processes.

The advantage of a meta-language like ConCurrent C is that it does not tie the program to any particular operating system. For example, UNIX¹ functions provide all of the mechanisms necessary for implementing all of the features discussed above, but direct use of its functions would make the code non-portable. Nonetheless, an MIMD program may be coded using the UNIX or other system calls directly. A meta-language like ConCurrent C is not absolutely crucial to MIMD code development for PASM; rather, it is a way to increase the portability of a program by shielding the user from the vagaries of the underlying operating system.

To demonstrate how a preprocessor for ConCurrent C would be built, the implementation of a few simple ConCurrent C constructs will be discussed assuming UNIX as an underlying operating system. Some study of the UNIX programming manual, Sections 2 and 3 [Ber83], reveals that each construct in ConCurrent C has a rough equivalent in a UNIX system function or group of functions. For example, consider the ConCurrent C *process* construct which has a syntax of the form:

process (*id*, *entrypoint*, *attributes*).

This causes the process identified by the variable *id*, which has its code at *entrypoint*, to be started. The optional *attributes* indicate where (what

¹ UNIX is a trademark of Bell Laboratories.

processor) the process is to execute and also system-dependent attributes such as the priority or environment. Its translated equivalent in standard C and UNIX would use *fork()* to create a child copy of the parent process, would place the process identifier returned by *fork()* into the variable *id*, and would then *exec()* the child into the process called for by *entrypoint*. Environment *attributes*, for example, would be passed to the child process via the *exec()* call as well. If there were additional *attributes* specifying that the new process was to be created on a processor other than the one on which the parent resides, all of these actions would be preceded by the UNIX system calls *socket()* (to define an endpoint of a communication channel), *bind()* (to bind a machine name to the socket), *connect()* (to try to establish the connection), *listen()* (to listen for pending connections), and *accept()* (to accept a connection). Once the connection is established, *send()* is used to send a message to the remote processor and *recv()* is executed by the remote processor to obtain the message. Interpretation of the message by the remote processor causes it to create the new process using the sequence of calls given earlier. It can be readily seen that the UNIX syscalls are often too primitive to be readily adopted by "normal" programmers. The preprocessor contains the "expert" programmer's knowledge of how to create processes on remote processors and emits the proper sequence of UNIX syscalls whenever the *process* construct appears.

Shared variables are another example of a ConCurrent C construct that is often used but is non-trivial to implement using UNIX syscalls. UNIX processes never share any user-oriented data. When a parent creates a child process, the child gets a copy of the parent's data but not access rights to the parent's data itself. Therefore, either processes must use UNIX files to hold shared variables or they must communicate with server processes that hold the shared variables. The first option is likely to have poor performance because before and after each transaction, it must be ensured that the in-core copy of a file is coherent with the disk copy. The second option is better but involves setting up special server processes to handle the transactions. Furthermore, communication with the server processes must be established using the socket-establishment and message-passing sequence given earlier. Again, this indicates that a meta-language implementing often-used MIMD constructs is highly desirable for shielding the user from the complexities of the operating system.

It also indicates that UNIX was not really designed for applications involving large numbers of co-operating sequential processes running on a set of homogeneous processors. Alternatives to UNIX as an underlying operating system for PASM are discussed in detail in a later chapter.

An alternative to a meta-language like ConCurrent C is simply a set of library functions that perform operations such as starting remote processes and manipulating shared data. This is a much "cheaper" alternative because a pre-processor need not be written. Its only drawback is that it affects the readability of programs. For example, consider declaring a shared variable *S* in ConCurrent C. The meta-language pre-processor can identify *S* as a shared variable each time it appears in an expression; for example,

```
if(S > 0),
```

and generate the appropriate sequence of operations to obtain its value. If the alternative scheme is used, the programmer is forced to remember that *S* is shared each time it is used. Even if the sequence of operations needed to obtain a shared variable's value was encapsulated in a library function, the programmer would still have to code something like

```
if(read_shared(&S) > 0)
```

to obtain the value. In short, a meta-language can perform some lexical and semantic transformations that the C compiler cannot and can therefore provide the programmer with a more understandable program.

It is not the author's intention to advocate ConCurrent C as the best source language for MIMD computation on PASM. There are certainly syntactical elements of ConCurrent C that the author finds objectionable and missing or mis-features of its semantics that limit its usefulness for a reconfigurable system such as PASM. Rather, ConCurrent C is presented as one possible set of useful constructs that are compatible with existing language methods and which can be developed independently of the SIMD language extensions.

An alternative to ConCurrent C is a language such as "Refined-C" [DiK85]. In this language, no explicit parallel constructs are added (e.g., process, cobegin) but data access rights are strictly defined by the programmer. This removes the barriers to automatic parallelism detection by the compiler. It also prevents users from making programming errors that result in "race"

conditions or which have unforeseen side-effects. These errors can occur when programming in ConCurrent C. The Refined-C implementor has extended a C compiler rather than used a translation approach because the C compiler is required to perform automatic parallelism detection. An alternative would be to translate Refined-C into an explicitly parallel language like ConCurrent C (or directly into conventional C with system calls) and leave the conventional C compiler unmodified.

2.4.5 Compilation Techniques for PASM

The PASM hardware is described in Part I of this thesis. Aspects of the PASM operating system are discussed in [TuS82a, TuS82b, TuS83, TuS84a, TuS84b, TuS85, KuS83] and later in Part II. The run-time mask stack is considered in [CIS83] and later in Part II. An assembler that produces object files for both serial and SIMD programs has been implemented and was described earlier and in Appendix A1.2.

Program Sections

The format of object files and the interpretation of the program sections was discussed earlier in Part II. Recall that when serial or MIMD programs are compiled, standard MC68000 assembly language is output. When assembled, three "sections" are created in the object file: text, data, and bss. These typically contain program code, initialized data, and uninitialized data, respectively. The complete object file is given to a processor (serial mode) or to a set of processors (MIMD mode).

When SIMD programs are compiled, SIMD MC68000 assembly language is output. When assembled, three subdivisions are created in the object file, one for the MC CPUs, one for the FBUs, and one for the VPEs. Each VPE gets a complete copy of the MIMD program code and $1/V$ th of the initialized data area (where V is the number of VPEs). Each VPE must also allocate and initialize a bss area.

The following classes of Parallel-C variables are assembled into the sections given in Table 2.4.1. Code that operates on scalar variables must be

Table 2.4.1. Program sections for Parallel-C variables.

| TYPE | SECTION |
|---------------------------|---------|
| initialized scalar | c_data |
| PE address mask constants | c_data |
| uninitialized scalars | c_bss |
| initialized selectors | p_data |
| uninitialized selectors | p_bss |
| initialized parallel | p_data |
| uninitialized parallel | p_bss |

executed by the MCs. Parallel variables must be manipulated by VPEs. Selectors are held by the VPEs but may be transmitted to the MCs to be used in mask operations. SIMD assembly language programs prefix each instruction with a "c_" or a "p_" to distinguish CU instructions from PE instructions.

Consider the following two functions:

```
double toFahr(x)
double x;{
    return(x * 9/5 + 32);
}
```

and

```
parallel [] double PtoFahr(x)
parallel [] double x;{
    return(x[*] * 9/5 + 32);
}
```

When the code for toFahr() is generated, the machine instructions will go into the c_text space and will have c_ prefixes. When the code for PtoFahr() is generated, the machine instructions will also go into the c_text space but will have p_ prefixes. Note that virtually the same machine instructions would be generated for both the serial and parallel functions (except for prefixes). This is because each PE performs the operations for only one data item just as the CU does. However, in the parallel case, the selector [*] will be evaluated to control the enabling of the VPEs.

Determining the Mapping of CPEs to VPEs

The PASM multistage interconnection network allows any source processor to communicate with any destination processor in one pass through the network stages. However, only some permutations (mappings from N source processors to N destination processors) can be performed in one "pass" through the network. The choice of a good mapping of CPEs to VPEs is based on an intuition about what types of permutations will be needed by algorithms.

Cube networks connect a number of processors that is a power of two. Also, PASM partitions always contain a number of PPEs that is a power of

two. This indicates that the total number of VPEs assumed for a program should be a power of two and be at least the minimum partition size: the use of any fewer does not represent a "savings" in any sense.

First consider the mapping used to convert one-dimensional CPE numbers into PASM VPE numbers. The number of VPEs dictated by a particular CPE extent is the length of the *smallest enclosing vector* of dimension one that has an extent that is a power of two. For example, if the declaration calls for the shape "parallel [12]," the smallest enclosing vector with a power-of-two extent has length 16. Therefore, if this were the only declaration, 16 VPEs would be assigned to this program. A number of typical operations on two vectors *a* and *b* of this size can be identified. One is aligned-element arithmetic or comparison operations as in:

$$a[*] + b[*].$$

However, since aligned-element computations do not involve the interconnection network, this should not constrain the choice of mappings. In another case, non-aligned elements might be operated upon, for example,

$$a[\text{evens}] + \text{shift_1D}(a[\text{odds}], 2).$$

Uniform shifts of this type can be performed by the PASM network in one pass. Therefore, the identity mapping is suitable for this type of operation. Finally, an operation of the form

$$a[*] + \text{rotate}(b[*], 3, 12)$$

might be used. Here, the network can support the uniform shift of distance 3 for CPEs [0]-[8] (data is sent to CPEs [3]-[11]) in one pass, but the modulo 12 shift of CPEs [9]-[11] (to CPEs [0]-[2]) takes an extra pass. The cube network can perform such a rotate operation only when the vector has a length that is a power of two. Use of a different mapping would not improve matters in this last case; only a network that allows modulo 12 shifts would be of use. Therefore, the identity mapping is the best (and most simple) that can be hoped for in the one-dimensional case.

For two-dimensional CPE numbers, consider a mapping based on the *smallest enclosing square* of VPEs that is a power of two. The shape declaration:

parallel [3][3]

is a perfect square but does not call for a number of VPEs that is a power of two. A variable with such a declaration would be embedded in the smallest enclosing square with size four-by-four. Again, the choice of embedding is influenced by the types of operations that can be envisioned for such a shape and how well the interconnection network would support them. Since the shape declaration implies a row-and-column addressing scheme, it is likely that the operations performed on such a variable will be row- and column-oriented. These might include row exchanges (e.g., matrix operations), row-wise and column-wise arithmetic (e.g., dot product calculation, matrix multiplication), or matrix transposition operations. Mapping the CPE numbers by embedding in the enclosing square ensures that each "column" is embedded in a power-of-two-sized vector and causes each new row to start on a power-of-two boundary. This allows row-exchange and other multiple-row operations to be performed by the cube network in a single pass.

Clearly, embedding in the smallest enclosing square can be very inefficient. Consider the embedding caused by the declaration

parallel [3][1000].

It would be undesirable (and is unnecessary) for a variable having this shape to be embedded in a 1024-by-1024 "square" of VPEs. A moment of thought will reveal that the rule for enclosing d-dimensional parallel shapes in d-cubes (d-dimensional figures) is more restrictive than necessary: a mapping with less waste of VPEs is obtained by padding all of the dimensions to sizes that are powers of two. Thus the declaration given above need be rounded only to a 4-by-1024 "rectangle" of VPEs with no loss of generality so far as cube network operations are concerned.

Now consider the VPE to LPE mapping performed by the operating system. PASM will assign VPE numbers to LPEs in a "modulo" fashion. That is, if there are V VPEs and Y LPEs, LPE y will simulate all VPEs v such that v modulo Y is y ($0 \leq v < V$, $0 \leq y < Y$). This ensures that if an interconnection network permutation is passable when VPE numbers are used to number the inputs and outputs of a "virtual network", the permutation will also be passable when LPE numbers are used and vice-versa. Another reason for the VPE

to LPE mapping is that if the number of LPEs is changed, the movement of VPE context information is straightforward. For example, if $Y=2$ and $V=8$, each LPE simulates four VPEs. If the operating system decides that the number of LPEs can be doubled ($Y=4$) (e.g., another job using those LPEs completes), each of the original LPEs off-loads two of the VPEs it is simulating. To maintain the "modulo" order, LPE 0 sends VPEs 2 and 6 to LPE 2 while LPE 1 sends VPEs 3 and 7 to LPE 3. This can be done in two parallel block transfers. If some other ordering was done; for example, the first V/Y VPEs in LPE 0 and the last V/Y VPEs in LPE 1, changing the number of LPEs would result in much more communication. Demonstrating this, LPE 0 would send VPEs 2 and 3 to LPE 1, LPE 1 would send VPEs 4 and 5 to LPE 2, and LPE 1 would also send VPEs 6 and 7 to LPE 3. This requires four block transfers rather than two. Similar behavior exists when the number of LPEs is halved.

Virtual MCs

Parallel-C presents the machine model of a variable number of PEs associated with a single control unit. Thus there is only one *conceptual CU*. In PASM, multiple control units (the MCs) may be assigned when the partition size chosen by the System Control Unit is larger than N/Q PEs. Therefore, a *Virtual MC (VMC)* can be defined to exist to control each virtual MC-group of N/Q VPEs. The System Control Unit determines how many *Logical MCs (LMCs)* to assign to execute the program and determines which *Physical MCs (PMCs)* are to be used. As discussed earlier, the number of LMCs actually assigned may be fewer than the number of VMCs called for due to system load, faulty hardware, or other factors. This implies that each LMC may be called upon to simulate the actions of multiple VMCs.

Consider a program requiring 32 VPEs that is to be run on the PASM prototype ($N=16$, $Q=4$). Here, V VMCs ($32/(N/Q)=8$) are required. Obviously, no more than four LMCs can be assigned to the program at any time; therefore, the Y LMCs assigned must multiplex their activity such that each simulates V/Y VMCs. General and PE address masks associated with this program assume that there are 32 VPEs. An LMC that is currently simulating VMC v

decodes masks and uses the bits corresponding to the VPEs to which it is connected (VPEs v, V+v, 2V+v, and 3V+v in this case). It also broadcasts instructions to its LPEs as would be normally done in SIMD mode. If two or more LMCs are assigned to the program execution, the VMCs are mapped to LMCs in the "modulo" fashion described earlier. Also, the operating system ensures that the VMCs are *co-scheduled* on the LMCs such that the first V/Y VMCs execute at the same time, then the next group of V/Y, and so on.

Each LMC contains a copy of the text section which is shared by all VMCs simulated by the LMC. The VMC data, bss, and stack areas are all independent within an LMC.

If a program does not contain any interconnection network operations or data-dependent operations (i.e., "if any"), the operating system can let the first V/Y VMCs execute to completion, then schedule and allow the next V/Y VMCs to execute to completion, etc. However, if these operations do occur, the PASM operating system ensures that VMCs are scheduled in the correct order. The following discussion describes how this order is maintained.

As described earlier, the SIMD library function `vpe_bcopy()` is used to perform VPE-to-VPE transfer operations. It consists of two syscalls: `write()` and `read()`, executed in that order. The two syscalls are separated by a synchronizing instruction executed by the VMC. Synchronization is necessary because all VPEs must `write()` before any can be allowed to `read()`. `Write()` takes data from the buffer specified by the user, determines on which PPE the VPE actually exists, sets the network, and transfers the data. Data arriving at a destination PPE is buffered by the local operating system tagged by the destination VPE number. (If the source and destination VPEs are being simulated by the same PPE, only a memory-to-memory copy is needed). As the synchronization point is reached by an VMC-group, a context switch is forced to begin simulation of another VMC-group. When all VMC-groups have reached the synchronization point, it is guaranteed that all VPEs have called `write()` and VMC-groups can be released to perform the `read()`. `Read()` takes data from the operating system buffer identified with a certain VPE number and copies it to the user's buffer area. A VMC-group performing `read()`s need not be swapped out until another network synchronizing instruction. The operating system will hold the data buffers for those VMC-groups that have not yet performed the

read() for as long as necessary. A similar scheme of scheduling VMCs in a round-robin order is used to coordinate "if-any"-type operations.

Indexing and Masking

When variables are indexed along the parallel dimension, the selection can be done with virtually any kind of constant or variable. The capabilities of the masking operations unit of PASM will influence how selectors are handled. Ideally, the masking operations unit would have a large number of registers for the storage of selectors. Recall that selectors can be initialized with general masks, PE address masks, or functions of PE address masks and are treated in expressions like ordinary parallel variables. For large machines with many PEs, the cost of moving selectors back to MC memory if there are an insufficient number of mask storage registers is very high. For machines with few masking operations unit registers, selectors should be compiled as "parallel [] char" and placed in the p_data or p_bss section. Placing selectors in the PEs imposes a small penalty at run time since the selector must be sent to the MC as a data conditional mask so that the MC can enable or disable the appropriate set of PEs. This penalty is exacted already when indexing parallel variables with other parallel variables.

There are four cases of assignment statements:

1. Left side scalar; right side scalar. This is the standard C assignment.
2. Left side scalar; right side parallel. The selector that indexes the right side is examined. If it has no true elements, the left side becomes undefined. If the selector has one and only one true element, that PE is enabled and is instructed to write to the MC/PE communications port. The MC can then retrieve the data and assign it to the left side. If the selector has more than one true element, the selector is modified such that it now has one and only one true element (which element is selected is undefined by the language and is implementation-dependent). The selected PE is then enabled to send the data as before.
3. Left side parallel; right side scalar. The MC places the selector of the left side into the Enable Signal Register. Then it constructs a "move immediate" instruction and writes it to the Immediate Broadcast Register. When

- the selected PEs receive and execute the instruction, the immediate value is assigned to the left side.
4. Left side parallel; right side parallel. After evaluating the right side, the MC places the selector of the left side into the Enable Signal Register. Then it broadcasts a "move" instruction to assign the value of the right side to the left side.

Compiling Control Flow Operations

As discussed in Section 2.4.3, control flow instructions are rather inefficiently executed in strict SIMD mode. Nonetheless, each control flow operation is shown here in both its strict SIMD mode version and in the SIMD/MIMD mode version. A number of "macro" operations are defined below to simplify the examples. Each of these operations is for notational convenience: in reality, they would be compiled as assembly language subroutines or in-line macro invocations. The "pusht <mask>" operation refers to and-ing the <mask> with the top of the conditional mask stack [CIS83] and pushing the result on the top of the stack. The "pushnt <mask>" operation refers to and-ing the <mask> with the next-to-top item of the conditional mask stack and pushing the result on the top of the stack. The "copy" operations pushes a copy of the value on the top of the conditional mask stack. The "pop" operation pops off the item on the top of the stack. The "none <mask>, <label>" operation tests the <mask> and branches to <label> if the mask has no true elements. DCM refers to the *Data Conditional Mask* that is formed by all PEs writing a true/false result to their Condition Code Register.

For the parallel *if* construct:

```

if ( <parallel-expr> )
    <statement1>;
else
    <statement2>;

```

Strict SIMD:

```

begin: p_xxx           ; evaluate parallel-expression
       p_mov.w    ccr,DCM   ; send to port
       pusht     !DCM      ; CU pushes complement for else-part
       pushnt    DCM      ; CU pushes for then-part
       <stmt1>          ; then-part
       pop        ; reset for else-part
       <stmt2>          ; else-part
       pop        ; reset

```

SIMD/MIMD

In c_text:

```
begin: p_bsr  iftest.
```

In p_text:

```

iftest:  xxx           ; evaluate parallel-expression
        beq      elsepart
        <stmt1>          ; then-part
        rts
elsepart: <stmt2>          ; else-part
        rts

```

For parallel *switch-case*:

```
switch( <parallel-expr> ){

    case <scalar-constant1>; <statement1>; break;

    case <scalar-constant2>; <statement2>;

    case <scalar-constant3>; <statement3>; break;

    ...

    default:           <default-statement>; break;
}
```

Note that any processor that executes <statement2> will also execute <statement3>. The most obvious way to compile this is:

```
if( <parallel-expr> == <scalar-constant1> )
    <statement1>;
else if( <parallel-expr> == <scalar-constant2> ){
    <statement2>;
    <statement3>;
}
else if( <parallel-expr> == <scalar-constant3> ){
    <statement3>;
}
else
    <default-statement>;
```

and use the previously defined method of handling the if-then-else statements. Unfortunately, this is not a very general solution since <statement3> is given twice (and thus would have to be broadcast twice).

The standard C compiler sorts the <scalar-constants> in ascending order and compiles in a binary search routine to identify the case to be executed at run time. Associated with each <scalar-constant> is a label that indicates the entry point for the statement block for that case. Cases that are terminated with "break" statements produce statement blocks that end with branches to

the instruction beyond the scope of the case statement. Cases that “fall through” have statement blocks without branches that “fall through” to the next case. This would be an appropriate approach for the SIMD/MIMD mode:

In c_text:

begin: p_bsr switchtest

In p_text:

```

switchtest:    xxx                                ; evaluate parallel-expression
                yyy                                ; binary sort to find match
                .
                .
                .
                .
                jmp      <ea>      ; computed goto on
                                ; entry point table

stmt1:         <stmt1>
                bra      endswitch

stmt2:         <stmt2>                            ; fall through

stmt3:         <stmt3>
                bra      endswitch

default:       <default-stmt>
                bra      endswitch

endswitch:     rts

```

The strict SIMD mode equivalent is much more complicated. Conceptually, a mask must be constructed for each statement block. A mask for `<statement1>` will have true elements for those processors where `<parallel-expr> == <scalar-constant1>` is true. The mask for `<statement2>` is derived similarly. For `<statement3>`, the mask is the “or” of the mask derived from testing `<parallel-expr> == <scalar-constant3>` and the `<statement2>` mask since PEs executing `<statement2>` also execute `<statement3>`. Finally, the `<default-statement>` mask is the complement of the “or” of all of the previous masks. Since the CU calculates a mask before broadcasting the instruction block, it can examine the mask with the “any()”

function to determine if any PEs will actually be executing instructions for the block. If not, the instructions need not be broadcast.

For strict SIMD:

| | | | |
|-------|----------------|---------------|----------------------------------|
| | p_xxx | | ; evaluate <parallel-expr1> |
| | p_mov.w | ccr,DCM | ; send to port |
| | c_or | DCM,defmask | ; keep running default mask |
| | none | DCM, L2 | |
| | pusht | DCM | ; push for <stmt1> |
| | <stmt1> | | ; execute it |
| | pop | | ; reset |
| L2: | p_xxx | | ; evaluate <parallel-expr2> |
| | p_mov.w | ccr,DCM | ; send to port |
| | c_or | DCM,defmask | ; update running default mask |
| | c_mov | DCM,lastmask | ; keep lastmask for fall-through |
| | none | DCM, L3 | |
| | pusht | DCM | ; push for <stmt2> |
| | <stmt2> | | ; execute it |
| | pop | | ; reset |
| L3: | p_xxx | | ; evaluate <parallel-expr3> |
| | p_mov.w | ccr,DCM | ; send to port |
| | c_or | DCM,defmask | ; update running default mask |
| | c_or | DCM,lastmask | ; accept fall-through |
| | none | lastmask, LD | |
| | pusht | lastmask | ; push for <stmt3> |
| | <stmt3> | | ; execute it |
| | pop | | ; reset |
| LD: | not | defmask | ; complement default mask |
| | none | defmask, Lend | |
| | pusht | defmask | ; push for <default-stmt> |
| | <default-stmt> | | ; execute it |
| | pop | | ; reset |
| Lend: | | | |

The parallel *while* statement:

```
while ( <parallel-expr> )
    <statement>;
```

is compiled in strict SIMD mode:

```
Lbegin: p_xxx           ; evaluate <parallel-expression1>
        p_mov.w   ccr,DCM   ; send to port
        none      DCM, Lend  ; done
        pusht     DCM       ; push for <statement>
        <stmt>          ; execute it
        pop          ; reset
        p_bra     Lbegin    ; go again
Lend:
```

For SIMD/MIMD:

In *c_text*:

```
begin: p_bsr   whiletest
```

In *p_text*:

```
whiletest: xxx           ; evaluate parallel expression
          bne      block   ; if true, execute statements
          rts          ; done
block:   <stmt>
          bra      whiletest ; try again
```

The parallel *do* and *for* are similar and are not given here.

Compiling User-defined Functions

As discussed in a previous section, functions with arguments that are of parallel type are treated specially. Specifically, parallel variables have a value and an implicit selector that indicates if a particular element of the parallel variable is currently defined. The top of the conditional mask stack represents the current enabled/disabled state of the processors. Only those processors that are enabled at the point of the function call participate in the evaluation of arguments, execution, and return from a function. A good example of a function that is only executed by some processors would be one that is within the "then" or "else" block of an *if-then-else* statement. For the purpose of discussion in this subsection, "all" processors is taken to mean "all enabled processors at the point of the function call."

Before a function is called, its arguments must be evaluated. Scalar arguments are treated normally (their values are placed on the stack). Parallel arguments must have both their values and their implied selector placed on the stack. During function execution, whenever a parallel variable is used, its implied selector is and-ed with the explicit selector or index that is applied to the parallel variable in the context of the function. Using the earlier example: if a function *f()* is called with a parameter *y* that has a selector *evens* as in

f(y[evens]),

within function *f()*, the corresponding formal parameter *spy* is given an implied selector "[evens]." No matter how *spy* is used within the function, no elements of *spy* for odd-numbered processors are defined. Thus, "*spy[*]*" can be interpreted as "*spy[* && evens]*" in this case.

2.4.6 Programming Examples

The programming examples briefly introduced below implement algorithms that have been run on the PASM simulator (to be described). Details about the algorithms are given in Part III of this thesis; therefore, a description of them is not duplicated here. Since no parallel-C compiler exists at this time, these programs were developed to test the ease of use of the language, not for actual compilation.

The first algorithm ("smooth.1D.c"; Appendix A3.1) smooths a TIrows-by-TIcols (TI = Total Image) image that is distributed equally among a machine logically configured as a \sqrt{N} -by- \sqrt{N} array of processors. N is a compile-time constant introduced to the compiler with a line such as:

```
cc -DN=4 file.c
```

which defines the name 'N' and its value '4' for the preprocessor. The program is assumed to be called with:

```
smooth TIrows TIcols < input
```

where it is assumed that TIrows/ \sqrt{N} and TIcols/ \sqrt{N} are integers. For simplicity, error checking is ignored.

In the second example ("smooth.2D.c"; Appendix A3.1), the smoothing program is given again, but the machine is assumed to have PErows -by- PEcols processors. PErows and PEcols are compile-time constants. Thus each PE will hold and process subimages that have dimensions TIrows/PErows -by- TIcols/PEcols. Note that the parallel arrays are given a two-dimensional shape. This avoids the use of the PE address masks which are difficult to manage when the machine sizes are not powers of two. Unfortunately, many of the functions in the libraries given earlier are defined only for a one-dimensional array of processors. Therefore, many of the arguments to the functions have to be cast into an appropriate type, making the code more difficult to read.

Another example ("histo.c"; Appendix A3.2) shows a histogramming algorithm defined for N processors (N is a compile-time constant). The length of the input that each PE processes is variable. It uses the interconnection network to perform recursive doubling operations to combine the results.

Finally, the last example ("egt.c"; Appendix A3.3) gives an Edge-Guided-Thresholding algorithm that is called with the same parameters as "smooth.1D.c." Note the use of the parallel *for* statement to perform different numbers of iterations in each VPE.

2.4.7 Syntax Summary

This summary of parallel-C syntax is intended more for aiding comprehension than as an exact statement of the language. It is based on the original C reference manual syntax summary in [KeR78].

Expressions

The basic expressions are:

```

expression:
    primary
    * expression
    & expression
    - expression
    ! expression
    ~ expression
    ++ lvalue
    -- lvalue
    lvalue ++
    lvalue --
    sizeof expression
    (type-name) expression
    expression binop expression
    expression ? expression : expression
    lvalue asgnop expression
    expression, expression

primary:
    identifier
    constant
    string
    PE-addr-mask
    (expression)
    primary(expression-listopt)
    primary[expression]
    primary[*]
    lvalue.identifier
    primary->identifier

lvalue:
    identifier
    primary[expression]
    primary[*]
    lvalue.identifier
    primary->identifier
    *expression
    (lvalue)

```

PE-addr-mask:
 [mask-specifier-list]

mask-specifier-list:
mask-specifier
mask-specifier-list mask-specifier

mask-specifier:
 {X10-string}
expression {X10-string}

The primary-expression operators

() || . ->

have highest priority and group left-to-right. The unary operators

* & - ! ~ + + - - sizeof(type-name)

have priority below the primary operators but higher than any binary operator, and group right-to-left. Binary operators and the conditional operator all group left-to-right and have priority decreasing as indicated.

binop:
 * / %
 + -
 >> <<
 <= >= <= >=
 == !=
 &
 |
 &&
 ?:

Assignment operators all have the same priority, and all group right-to-left.

asgnop:
 = += -= *= /= %=
 >>= <<= &= ^= |=

The comma operator has the lowest priority, and groups left-to-right.

Declarations

```

declaration:
    decl-specifiers init-declarator-listopt ;

decl-specifiers:
    type-specifier decl-specifiersopt
    sc-specifier decl-specifiersopt
    sac-specifier decl-specifiersopt

sc-specifier:
    auto
    static
    extern
    register
    typedef

type-specifier:
    char
    short
    int
    long
    unsigned
    float
    double
    struct-or-union specifier
    typedef-name

sac-specifier:
    scalar
    parallel shape-declarator

init-declarator-list:
    init-declarator
    init-declarator, init-declarator-list

init-declarator:
    declarator initializeropt

declarator:
    identifier
    (declarator)
    *declarator
    declarator()
    declarator[constant-expressionopt]

shape-declarator:
    shape-declarator[constant-expressionopt]

struct-or-union-specifier:
    struct {struct-decl-list}

```

```

struct identifier { struct-decl-list }
struct identifier
union { struct-decl-list }
union identifier { struct-decl-list }
union identifier

struct-decl-list:
  struct-declaration
  struct-declaration struct-decl-list

struct-declaration:
  type-specifier struct-declarator-list ;

struct-declarator-list:
  struct-declarator
  struct-declarator, struct-declarator-list

struct-declarator:
  declarator
  declarator : constant-expression
  : constant-expression

initializer:
  = expression
  = { initializer-list }
  = { initializer-list, }

initializer-list:
  expression
  initializer-list, initializer-list
  { initializer-list }

type-name:
  type-or-sac-specifiers abstract-declarator

type-or-sac-specifiers:
  sac-specifier
  type-specifier
  sac-specifier type-specifier
  type-specifier sac-specifier

abstract-declarator
  empty
  ( abstract-declarator )
  * abstract-declarator
  abstract-declarator()
  abstract-declarator[ constant-expressionopt ]

typedef-name:
  identifier

```

Statements

compound-statement:
 { declaration-list_{opt} statement-list_{opt} }

declaration-list:
 declaration
 declaration declaration-list

statement-list:
 statement
 statement statement-list

statement:
 compound-statement
 expression;
 if(expression)statement
 if(expression)statement else statement
 while(expression)statement
 do statement while(expression);
 for(expression-1_{opt}; expression-2_{opt}; expression-3_{opt})statement
 switch(expression)statement
 case constant-expression: statement
 default: statement
 break;
 continue;
 return;
 return expression;
 goto identifier;
 identifier. statement
 ;

External definitions

program:
external-definition
external-definition program

external-definition:
function-definition
data-definition

function-definition:
 $\text{type-or-sac-specifier}_{\text{opt}} \text{ function-declarator function-body}$

function-declarator:
 $\text{declarator} (\text{parameter-list}_{\text{opt}})$

parameter-list:
identifier
identifier, parameter-list

function-body:
 $\text{type-decl-list function-statement}$

function-statement:
 $\{\text{declaration-list}_{\text{opt}} \text{ statement-list}\}$

data-definition:
 $\text{extern}_{\text{opt}} \text{ type-or-sac-specifier}_{\text{opt}} \text{ init-declarator-list}_{\text{opt}};$
 $\text{static}_{\text{opt}} \text{ type-or-sac-specifier}_{\text{opt}} \text{ init-declarator-list}_{\text{opt}};$

Preprocessor

```
#define identifier token-string
#define identifier(identifier, ..., identifier) token-string
#define undef identifier
#define include "filename"
#define include <filename>
#define if constant-expression
#define ifdef identifier
#define ifndef identifier
#define else
#define endif
#define line constant identifier
```

2.4.8 Summary

A superset of the C programming language that is applicable to the SIMD/MIMD mode processing environment of PASM has been described. The language extensions proposed for SIMD mode are quite typical of those introduced in other SIMD mode languages: parallel variables, indexing along the parallel dimension, definition of data alignment functions, etc. Others such as multi-dimensional parallelism and the notion of a conceptual PE are unique to Parallel-C. Compilation techniques for PASM are novel due to the machine's ability to reconfigure as virtual SIMD or MIMD machines with varying number of processors. The extensions for MIMD mode are much more operating-system-related than language-related: a meta-language approach similar to ConCurrent C was proposed. Such an approach allows the development of the compiler for the SIMD extensions and the translator for the MIMD extensions can proceed independently.

CHAPTER 5

PASM OPERATING SYSTEM STUDIES

2.5.1 Introduction

This chapter discusses the subsystems that manage the data and program files and schedule and monitor tasks within the PASM system. Section 2.5.2 makes a case for why the *Memory Management System (MMS)* of PASM requires a distributed approach. The handling of large data sets and the distribution of these files among MSUs is discussed in Sections 2.5.3 and 2.5.4. The interaction of the MMS processors is considered in 2.5.5 and a job scenario is outlined. Parts of this chapter were published in [KuS83]. Section 2.5.6 overviews the work done in organizing the hierarchy of the PASM prototype operating system.

Definitions of a number of operating system terms are appropriate at this point. A *user* is an individual that has access to PASM. The individual has a unique *login* name that he uses to identify himself. During a *login session*, the user interacts with the operating system. The operating system stores arbitrary-length collections of bytes in *files* identified by *filenames* that are associated with a single user. Files are arranged in a hierarchical structure similar to UNIX. *Superuser* is a user with special privileges who maintains system integrity and security.

A *process* is a single instruction stream (object file) being executed by a processor. It is assigned a *process identifier (pid)* (an integer) when it is created. An *SIMD process* is a process that runs on one or more MCs and their PEs. An *MIMD process* is a process that runs on a single PE. Any process may *fork* to create an identical copy of itself. If an MC forks an SIMD process, an identical SIMD process is created. Forking an MIMD process creates an identical MIMD process. If PEs in SIMD mode fork, they all do so

in parallel to form a number of MIMD processes. MIMD processes created in this way must eventually *join* after which control reverts to the SIMD parent. Processes that belong to different users are prohibited from interfering with each other.

A *task* is one or more SIMD and/or MIMD processes descended from a single ancestor. When users start a program running by executing its object file, a single process is begun; hence, at the start of execution, the task involves one process. If the process forks, the operating system creates more processes to run the task. Several tasks (multiple object files) may be run in parallel by arranging them in a *pipeline* so that the output of one feeds the input of the next (i.e., UNIX pipe). Together, the set of tasks constitute a *job*.

2.5.2 Memory Management System Overview

The design of the PASM Memory Management System is complicated by the fact that data files are distributed over multiple secondary storage devices thus requiring a parallel file system strategy. Files may also have to be broken down into subfiles which are handled independently because of processor memory size restrictions. A distributed processing approach to memory management is taken, using four interacting dedicated microprocessors. The memory management techniques presented here for PASM can be adapted for use in other large-scale systems.

The MMS controls the loading and unloading of the PE memory modules and employs a set of cooperating dedicated microprocessors. These are the Directory Processor (DP), Memory Scheduling Processor (MSP), Command Distribution Processor (CDP), and the Input/Output Processor (IOP). Figure 1.4.2 shows the interconnection of these processors.

The Directory Processor receives requests from the SCU and MCs to load or unload PE memory modules. In turn, it generates commands for and coordinates the actions of the other MMS processors. The Memory Scheduling Processor receives the commands from the Directory Processor and determines the order in which they should be performed. The Command Distribution Processor issues these commands to the MSUs and processes the acknowledgement of a command's completion. The Input/Output Processor handles the transfer of

files between the MSUs and peripheral devices. It also coordinates the reformatting and distribution of files among MSUs.

This distributed processing approach is chosen in order to provide the MMS with much processing power at low cost. A large amount of power is required since it is not desirable to burden the SCU with any memory management tasks. Furthermore, the number of files to be managed is enormous: a user's request for an input file for a 1024-PE SIMD program would involve the management of 1024 file directory lookups and transfers. The management problem becomes more severe when multiple simultaneous users of PASM are considered. The distributed approach to memory management will allow operations such as directory lookup, scheduling, and communication with the MSUs to be handled simultaneously. In addition, dedicating specific microprocessors to certain tasks simplifies both the hardware and software required to perform each task.

Digitized images used in image processing and pattern recognition algorithms consume prodigious amounts of memory. Image sizes of 1024-by-1024 pixels are typical, and may be as large as 5000-by-5000 pixels for some mapping applications [MiR81]. Thus efficient methods of segmenting large images, distributing the segments among as many as 1024 processors, and providing secondary storage for image data files must be considered so that the primary/secondary memory communication system of PASM does not become a bottleneck.

The MMS makes use of the double-buffered arrangement of memory modules to enhance system throughput. Thus while a processor is using one of its memory units, the MMS can be unloading the results of a previous task from the other unit and then loading it for the next task. When the current task is completed, the PE can switch to the other memory unit and begin the next task.

Ideally, all of the data for a task would be loaded into the appropriate memory units before execution begins. However, this would require the user to specify the files to be loaded in a way apart from the object file to be executed. Some new method would have to be developed to communicate to the SCU that "file X needs to be loaded whenever object file Y is to be run." No PASM proposal describing a general technique for specifying file names to be loaded

has ever appeared. It has been suggested that the SCU handle file operations before execution begins (to pre-load program and data files) and after it ends (to unload processed data files) and that MCs handle file operations during execution. Fundamental questions arise with such a technique: if there are multiple files used during the course of execution, how many should the SCU pre-load? How many should the MCs be responsible for? What if the first file to be loaded is dependent on an execution-time user input or if there are an unspecified number of files (e.g., load all files matching a "pattern")? Where in the MC and PE memories should the programs and data be placed? Will data placed in PE memories conflict with "local variable" areas being used by the active process? The approach of sharing control between the SCU and MCs for file operations and placement is unwieldy and in the author's opinion, ill-conceived.

In conventional computer systems, file I/O operations are specified within the object file itself: the process is scheduled, execution begins, and file operations cause the process to *block* (be suspended) until the file operation is complete. There is no ambiguity about what is to be loaded or unloaded: file operations are explicitly specified at the time they are to be done. Nothing is pre-loaded by the scheduler and nothing is unloaded automatically. Why was this approach never considered in earlier PASM writings?

The author's view is that the technique of allowing SIMD and MIMD processes to block for I/O operations was considered by others to be wasteful because the MCs and PEs could not be utilized for other purposes while they were waiting for the I/O operation to complete. This was based on the assumption that MCs and PEs, once started on a process, would run to completion and never switch context during processing. Context switching *preempts* the current process in favor of another one that is ready to run. Never performing a context switches is a *non-preemptive* strategy. Technically, the non-preemptive strategy is useful only for analysis purposes or for dedicated or real-time computer systems. PASM is neither a dedicated nor a real-time system. If PASM does not preempt processes, a careless or malicious user could easily deny access to the machine to other deserving users by running processes that never terminate. Also, non-preemptive strategies can be unfair. Consider a one-second duration task needing one MC-group "caught" in the run queue

behind a 1-day duration task currently using all of the MC groups. If a priority system specifically requests that the 1-day task has higher priority, so be it. The use of priorities implies sound reasoning behind the ordering of tasks; this is not the case for non-preemptive scheduling.

In conventional computer systems, a context switch occurs either because a process is blocked (due to I/O or some synchronization operation) or because its time slice has expired. It is readily acknowledged that the time slice for PASM should be chosen to be much longer than for conventional computer systems, perhaps tens of seconds as compared to tens of milliseconds. PASM context switches are more expensive than those in conventional systems because they involve coordination among multiple processors. With this caveat, no further barriers or objections to multi-tasking MCs and PEs can be raised. Preemption does not necessarily imply that the former process must be "rolled out" to a disk unit. Because the prototype MC and PE memories are large and can be managed by the associated CPU, more than one process in memory can be ready-to-run. Nor does the scheme of deferring file loading to execution time imply that files will be loaded less efficiently: if one memory unit contains the blocked process while the other memory unit contains a second process with data that is ready-to-run, there will be no memory access conflict while executing the second process. Later, if the second process blocks, data for the first may have already arrived making that process ready-to-run.

The file loading policy can be a great deal more flexible at execution time as compared to pre-execution time. Operating system calls can be provided that give the programmer more "say" in suggesting how much of each file should be loaded initially, when the file is no longer needed, and what guidelines to use for partitioning memory. If desired, a "preload" system call could be implemented to inform the operating system in advance of when a file will be needed. In short, the programmer, not the operating system, knows the characteristics of a program; system calls associated with the program itself are the best way of communicating the user's desires to the operating system in a consistent and timely manner.

There are two situations where the SCU does have to get involved in communicating file names and other information to the MCs. Some file names are not embedded in the object file (e.g., UNIX redirection of standard input and

output) and need to be communicated to the MCs. Also, tasks connected in a pipeline may be running on multiple partitions. The unnamed pipes that connect each pair of tasks must buffer their data somewhere; this is likely to be in the input queues of PEs associated with the end of each pipe. Here, the MCs need knowledge of where (which MC group) the end of a pipe is connected so that the network can be used to transfer the data across partition boundaries. Because both pipes and redirection can be initialized and/or canceled from within programs, these needs still do not legitimize the SCU preload/unload approach.

2.5.3 Handling Large Data Sets

There may be cases where not all of the data will fit in the single memory unit of all of the PEs executing the task. Assume that a *memory frame* is a block of data from secondary storage that is to be loaded at one time into the PE memory units associated with a given task. There are tasks where many memory frames are to be processed by the same program (e.g., maximum likelihood classification of multiple independent satellite pictures). The double-buffered memories can be used so that as soon as the data in one memory unit is processed, the processor can switch to the other unit (next frame) and continue executing the program. Each memory frame is composed of up to MN/Q files, where each file is the portion of a memory frame that is associated with an individual PE in the partition. Note that a memory frame may be composed of fewer than MN/Q files since some PEs might not require the data in a certain frame. This situation might occur if certain PEs are temporarily disabled, and thus do not participate in processing a particular frame. Note also that the files of a given frame need not be all of the same size. This situation occurs if the data is unevenly distributed among the PEs (i.e., an image file that has a number of pixels that is not divisible by the number of PEs).

When multiple memory frames are used, there must be a mechanism that allows the variable length portions of program or data sets (*local data*) stored in one memory unit to be made available to the other unit when the processor switches units to access the next memory frame. Alternatives for allowing access to local data were discussed in Part I, Chapter 6.

2.5.4 Distributing Files Among Storage Devices

The previous section described the hierarchy of image data files: image data is composed of one or more memory frames, and memory frames are composed of up to MN/Q files. This section will consider the file naming and placement scheme for the PASM memory system.

Consider an image data file logically configured as R rows and C columns of pixels (picture elements). For simplicity, assume $R = C = N$ and that each pixel is one word long. Some typical *formats* (ways of distributing) this data among N PEs are:

1. Subarray per PE: the original image is divided into N subimages, each $\sqrt{N} \times \sqrt{N}$ pixels. PE 0 gets rows 0 to $\sqrt{N}-1$ of columns 0 to $\sqrt{N}-1$, PE 1 gets rows 0 to $\sqrt{N}-1$ of columns \sqrt{N} to $2\sqrt{N}-1$, ..., and PE $N-1$ gets rows $(\sqrt{N}-1)(\sqrt{N})$ to $N-1$ of columns $(\sqrt{N}-1)(\sqrt{N})$ to $N-1$.
2. Row per PE: the original image is divided such that PE i gets row i .
3. Column per PE: the original image is divided such that PE i gets column i .

Each subimage (subarray, row, or column) is associated with one and only one PE, and may be comprised of one or more files. Recall that parts of the subimage may be loaded into PE memories at different times when changing memory frames. It was shown earlier that each PE is serviced by one and only one MSU. Further it was shown that only those PEs that could serve as logical PE i were connected to MSU j , where $j = [i/M]$. Therefore, the files comprising subimage i can be stored in MSU j without fear that they will ever need to be moved to another MSU, regardless of the partition on which a task is run. Also, this placement policy is independent of the physical partition size that is chosen at execution time.

With such a large number of data files, some file naming convention is required. The file names will be designated according to the principles of the UNIX [Ker79] operating system since this approach is familiar to most readers. The proposed convention for file naming is:

user/data_file/logical_PE_number/frame_number.format

For example, a user named "jane" might create an input data file comprised of two frames of row-formatted data called "picture." If her algorithm was written for execution by eight VPEs, the following files would be created from her

original data file, assuming $Q=4$:

| | |
|----------------------|-----------------------------|
| jane/picture/0/0.row | Files to be stored in MSU 0 |
| jane/picture/0/1.row | |
| jane/picture/1/0.row | |
| jane/picture/1/1.row | |
| jane/picture/2/0.row | Files to be stored in MSU 1 |
| jane/picture/2/1.row | |
| jane/picture/3/0.row | |
| jane/picture/3/1.row | |
| jane/picture/4/0.row | Files to be stored in MSU 2 |
| jane/picture/4/1.row | |
| jane/picture/5/0.row | |
| jane/picture/5/1.row | |
| jane/picture/6/0.row | Files to be stored in MSU 3 |
| jane/picture/6/1.row | |
| jane/picture/7/0.row | |
| jane/picture/7/1.row | |

In general, FMN/Q files are created, where F is the number of frames. Each MSU is responsible for storing MF of these files. A directory for every user of PASM is found on each MSU disk. User "jane" has a subdirectory on every MSU disk named "picture" which is the name by which she knows and refers to the file. On a particular disk, MSU 0 for example, the directory "picture" has two subdirectories, 0 and 1, in which the data corresponding to logical PEs 0 and 1 is stored. Since there are two files associated with each logical PE, each of these directories contains two files, one for each frame.

The approach of a fixed name convention for data files is desirable since it requires the user to remember only the file name ("picture") and the format ("row") which is desired. Therefore, a program needs only to communicate the name and format to the MMS. This is preferable to the communication of up to FMN/Q file names. The "format" is a user-defined naming convention. The file system does not attach any significance to the "format" whatsoever.

MIMD tasks will require that both the MIMD programs and associated data files be stored in the MSUs. While the MIMD environment does not lend

itself so well to fixed file naming conventions, there are less rigid rules for assigning MIMD processes to PEs. Thus if the programs for two MIMD processes A and B exist on MSUs 0 and 1, respectively, the A process should be executed on a logical PE 0 and B on a logical PE 1 if possible. Clever scheduling algorithms will be able to resolve most conflicts; however, occasionally a program and data file may have to be moved from an MSU i to an MSU j.

If an MC-group is available, it could be pressed into service to perform the file movement: the PE connected to MSU i would get data from the disk, transfer it through the interconnection network to the memory of PE j, where it would be available to MSU j. Such a menial task as transferring data from disk to disk might be better and more efficiently handled at the MMS level using inter-MSU channels. Experience with actual PASM usage patterns is needed to determine how often this situation would occur and its effect on system throughput.

It is the author's experience that an MIMD program would be generally be written to be compiled and link-loaded into a single object file. This is because the constituent processes are initially created from a single parent and may share synchronization variables. Such a program run on PASM would have identical copies of the object file on all MSUs. When processes do create new processes outside the program, e.g., via UNIX fork() and exec() calls, the new process is very often one that is "standard" for an operating system and would be expected to be stored on all MSUs. Examples of this type are calls to system sorting routines, stream editors, or command processors.

2.5.5 Memory Management System Processors

As discussed earlier, there are four MMS processors: the Directory Processor, Memory Scheduling Processor, Command Distribution Processor, and the Input/Output Processor. A "scenario" approach will be used to describe the interactions of these processors with the MSUs, SCU, and MCs.

In SIMD mode, the SCU uses scheduling algorithms to determine the scheduling order of tasks to be executed on the parallel machine. Since a task might be run on only a subset of the N PEs, it also determines the partition on which the task is to be run. The partition is fully described by two quantities:

the designator, and M, the number of MC-groups required by the task. Operating system issues such as scheduling and the schema for preloading memories are discussed in [TuS82b, TuS83, TuS84a, TuS84b, TuS85]. The descriptions given here are not in conflict with the scheduling algorithms but the memory preloading scheme has been largely dismissed with the arguments given above.

When a task is scheduled, the SCU informs the MCs in the partition that has been scheduled the object file name of the program, the designator, the number M, and the process id number. The command line that the user generated to start the process is also passed. This is because it may contain command-line arguments needed by the executable object file, names of files for input and output redirection, and so on. The designator MC is placed in charge of the partition for the purposes of that task.

Since each MC is in charge of managing its own memory, it must inform Control Storage where the program is to be made resident. With only four MCs in the prototype, each MC can communicate with Control Storage individually without inducing much overhead. With many MCs, a better solution would be to reserve a small amount of MC memory as a message exchange area and to let the designator MC tell Control Storage to check these areas for I/O "work" to be performed. Which MCs' areas to check can be specified by the designator and M.

It is the role of the Control Storage CPU to interpret object file headers and to split the object file up into its MC CPU and FBU components. Then, Control Storage makes the four-part object file headers and the MC CPU and FBU portions of the object file resident in the MCs. The sizes of the sections described in the header are used by the MC CPU for initializing the bss area, setting memory management registers if needed, determining how many VMCs it is simulating, and so on.

Part of the MC CPU's initializing process involves preparing the PEs for the loading of their part of the object file. Because PEs manage their own memories as do MCs, each PE is directed to place in a specially-designated area of its memory the address where the object file is to be made resident. The designator MC then informs the Directory Processor that PE portions of the object files should be made resident. The MC passes the file name, designator,

M, the process id, and optionally, a "priority" indicating the relative importance of this operation.

The Directory Processor was originally conceived to maintain a "file directory listing" of the MSU's contents. It was to use this information to check if the file exists. (The Directory Processor and the set of MSUs would communicate at system start-up time to initialize the Directory Processor's file name table.) Specifically, the file name table was to include the names of program and data files. File locations (MSU number) were also to be kept for MIMD files that were not replicated across the MSUs. In retrospect, this processor is probably superfluous and ill-conceived. One of the difficulties is the coherence problem that could exist between the file system maintained by the MSUs and that of the Directory Processor. Another is that the Directory Processor does not "do" anything other than to detect some errors earlier than the MSUs could and to associate an MSU number with a non-replicated file. The early error detection is probably not worth the programming effort because not all errors can be detected at this stage. The MSU location problem can be addressed by either replicating all program and data files intended for MIMD mode (simple but possibly expensive) or by placing "symbolic links" in MSUs that lack a non-replicated file. The symbolic link would contain the number of the MSU that actually has the file. The link could be a special type of disk file as it is in UNIX or could be placed in an MSU's primary memory. It would take only one "probe" to determine the MSU number that holds a file: if the "guessed" MSU was wrong, the correct one would be immediately identified. This MSU "probe" is likely to be no more costly than one performed at the Directory Processor; therefore, there is little reason to "probe" at a Directory Processor. For these reasons, the Directory Processor will be ignored in the following scenario discussions.

The Memory Scheduling Processor maintains a queue of requests to load and unload data files. Each request type has its own queue. A load request is a request for a program object file or input data for a task which is blocked by this request but otherwise ready-to-run. An unload request is a request to write processed output data to disk. Such a request may be generated before the task actually completes; for example, once the output file is "closed" by the controlling program, it is no longer needed and may be unloaded. Load

requests have a higher priority since the MC(s) and PEs might be blocked if they are waiting for input data. The choice between object file and data file load requests should be based on which request can be satisfied the fastest. Always ensuring that something is ready-to-run is the key to high utilization. Unload requests have lower priority than load requests since the associated task has already completed execution and the only resource being idled is the "user" who is awaiting results. Even though load requests have the higher priority, an unload request from a previous task might have to be scheduled first to allow the load request to be satisfied.

The Memory Scheduling Processor monitors the progress of each type of loading request. In SIMD mode, each type of request causes all of the MSU's to become busy, regardless of the size or designator of the partition. If some MSU's finish earlier than others, they may begin work on the next request as determined by the Memory Scheduling Processor. The Memory Scheduling Processor makes use of the "priority" to order requests *within* the loading and unloading request queues. Information in the Memory Scheduling Processor queues may also be updated by sending a message indicating a new "priority."

Note that the SCU determines where tasks execute but not the order of their execution. Consider two tasks arriving at the SCU both requesting the full machine as the partition size. The first has a small object file and a large data file while the second has a small object file and a small data file. Suppose that Control Storage and the MSU's take one second to make the small files resident and ten seconds for the large file. Obviously, the SCU is unaware of the difference in the data file sizes. Even if the second task arrives at the SCU several seconds later than the first, it is advantageous to load its small data file and to begin executing it. Execution of the first task cannot begin until the large data file is loaded; therefore, the MCs and PEs would be wasted for this period. By delaying the first task's large data file loading by one second, the MCs and PEs can be allowed to run the second task, possibly even completing it by the time the first task's data is loaded. Allowing the SCU to strictly determine execution order is undesirable when equal-priority tasks need service because it may lead to poor utilization as evidenced by the example. If necessary, priorities can be used to strictly control execution order.

When the Memory Scheduling Processor has determined the loading request that will be done next, it passes to the Command Distribution Processor the filename, descriptor, M, the process id, and the type of operation to be performed (i.e., load or unload). The Command Distribution Processor coordinates the actions of all of the MSUs, instructing each what files are to be read/written from/to each PE memory. Returning to the scenario, the Command Distribution Processor would direct each of the MSUs to retrieve the PE part of the SIMD program file, to examine the message areas of the PEs in the selected partition to determine where to load the object file, and finally to make the PE part of the object file resident.

As discussed in Part I, each MSU has its own CPU and memory which are used to manage the file system on the physical disk. The MSU probes the disk blocks and follows the directory structure of the file system until the requested file is found.

As each MSU completes the loading/unloading of a file, it reports back to the Command Distribution Processor to obtain more "work." When the Command Distribution Processor runs out of "work" associated with a given loading request, it informs the Memory Scheduling Processor. The Memory Scheduling Processor provides the Command Distribution Processor with the next request so that no MSU will be idled. When a loading/unloading request is completed, the Command Distribution Processor informs the Memory Scheduling Processor so that it can remove the loading request from the queue. The request completion "acknowledgement" is passed back along the chain of processors to its source: MSU, Command Distribution Processor, Memory Scheduling Processor, and MC.

Once the program is loaded in both MCs and PEs and initializations are performed, execution can begin. Execution can proceed until either MC data or PE data files are needed. (It is possible that neither will be needed). If the process becomes blocked due to an I/O request and if there is another task ready-to-run, the MCs and PEs switch context and execute the other task.

Requests for data file loading/unloading operations can be initiated by either the MCs or the PEs. For example, if all PEs in SIMD mode need the same file, the request can logically be initiated by the MC as part of the SIMD control program. If there are multiple memory frames and one or more PEs

wishes to have an alteration in the loading sequence, the requests are initiated on a per-PE basis via the controlling MC. (An alteration of the loading sequence may be desired if the data file to be analyzed next depends on the result of processing a previous data file.) PEs communicate requests to the MSU's via their MC and the MMS. While this may seem to be a longer path of communication, only the MMS knows the "global" state of the Memory System and can determine the priority of a request.

The MSU provides the PE with status information concerning the loading/unloading of a file. This status information is written by the MSU into the mutually agreed-upon PE memory area. The information indicates the type of transaction that took place, the file name, the number of bytes read/written, and the process id. This information is checked by individual PEs before they proceed to execute to ensure that the proper data file(s) were loaded or that data has been safely unloaded and the memory space made available for subsequent use.

When data files are to be loaded into the PEs, it is most convenient for each PE to generate the name of the file it requires and to place the name and the address where it should be loaded in the designated I/O "work" area. PEs request service via their MC which forwards the request to the MMS and then on to the MSUs. In order to identify to the MMS which PEs need service, each MC can compose a bit vector which has bit i non-zero for the i 'th logical PE associated with it. This aids the Command Distribution Processor in determining which MSUs need to be directed to examine PE memories. An alternative to the bit-vector scheme which would reduce traffic on the MC-MMS-MSU links would simply be to direct every MSU to examine all of the PE work areas for pending "work." PEs would set a flag indicating that work to be done was present in the I/O "work" area and MSUs would examine the flag, perform the work, and negate the flag.

The Input/Output Processor is the MMS interface to devices such as graphics terminals, tape drives, or hard copy output devices. This processor is responsible for reformatting data files for distribution among MSUs. For example, the "picture" file read in from a tape might have to be reformatted such that each "row" is distributed to a different file. The Input/Output Processor is also involved in the "piecing together" of data files distributed among the

MSUs to produce a single file that may be displayed on a graphics terminal or printed on a hard copy output device. A high-speed data link connects the Input/Output Processor with the SCU (to which the user's terminal is connected). This link will allow processed data (e.g., images) to be sent to the SCU and displayed on the user's terminal efficiently without disrupting normal communications between the SCU and MMS.

As discussed earlier, MIMD programs and data files occasionally will need to be migrated from MSU to MSU. The Input/Output Processor is responsible for initiating and controlling these data transfers.

2.5.6 PASM Prototype Operating System Hierarchy

Because the equipment grant for the PASM prototype is almost exclusively dedicated to hardware development, funds for commercial software are limited. Also, due to the licensing agreements between Purdue University and the vendors of VAX/UNIX software, much useful software is available to the PASM Parallel Processing Laboratory, but may only be run on the local VAX computers. This includes utility programs such as compilers and linker-loaders, operating systems such as UNIX, and library routines (e.g., mathematical functions) distributed with UNIX 4.2 BSD. While the utility programs can be modified so that they act as cross-development tools, no licensed software may be run on PASM CPUs.

A serial C-language to MC68000 cross-compiler is already developed and operational on the local VAX/UNIX machines. There is no difficulty with its use for PASM as far as the VAX/UNIX licensing agreement is concerned as long as the software is executed on the VAX and the resulting object code is downloaded into PASM.

The parallel extensions to the C-language compiler (Parallel-C) are being made by the author. (The implementation and demonstration of an operational Parallel-C compiler is not a part of this thesis work.) Extensions made to the compiler thus far cannot be documented here (by listing the modified code) because of licensing restrictions. If and when the Parallel-C compiler is completed, it must also operate in a cross-compiled mode because it is wholly derived from licensed software.

The extension of the MC68000 cross-assembler to handle parallel programming constructs and more advanced members of the 68000 family has been completed and was described in an earlier chapter. The cross-assembler is in the public domain. Although the assembler could be run in "native" PASM mode, it calls UNIX system library functions to perform I/O. Therefore, it cannot be transported to PASM unless a PASM CPU is licensed for UNIX.

Consider then the licensing of one or more PASM CPUs to run UNIX. The positive aspects of acquiring a UNIX license are that:

1. it would allow PASM to self-compile its programs and would eliminate the time needed for downloading operations;
2. it would keep the development team from having to "re-invent the wheel" each time a utility program was to be provided on PASM;
3. its extensive set of development and debugging tools could be used without having to move programs and data between machines;
4. users could be one step closer to the PASM machine, reducing dependence on machines not maintained by the PASM laboratory team.

The negative aspects of acquiring an operating system such as UNIX for the PASM prototype are that:

1. the current prototype CPU boards do not have the proper memory management facilities on them and would have to be replaced;
2. UNIX needs at least two large disks (85 M-byte or more) to provide a reasonable environment to support multiple users;
3. UNIX is large and therefore difficult to modify or enhance, particularly by students;
4. UNIX is an inappropriate operating system in its raw form for parallel processing systems, although it would be of use as a host operating system in the SCU.

Because of these facts and the cost of purchasing suitable hardware that would support a multi-user PASM host development system, the decision was made to continue to use the cross-development tools provided by the local Engineering Computer Network VAXen. Because *some* operating system was needed to implement communication among processors, process scheduling, and a distributed file system, work began to develop an in-house system that would use public-domain software whenever possible. One such public-domain

operating system is XINU [Com84] which was developed by Douglas Comer of the Purdue Computer Science Department. It is currently being studied and modified for use on all PASM CPUs.

The XINU operating system is designed for simple hardware that may or may not have a disk system. Thus it is ideal for the PASM environment especially for providing a kernel that would run on each processor in the system. Each PE, MC, Memory Management System processor, as well as the SCU and processors associated with the secondary storage devices would run the same version of the basic kernel. Each type of system component would have some system-oriented routines that implement the particular functions it performs. For example, the SCU would have parallel scheduling algorithms, the secondary storage device processors would have specialized file management routines, etc. In SIMD mode, the kernel is not active in the PEs except if some exception (e.g., bus error, divide by zero) occurs or if the PE temporarily enters MIMD mode. Otherwise, the kernel plays a vital part of handling the resources local to each processor.

XINU is a layered operating system for a uniprocessor. It implements a process manager and process coordination primitives such as semaphores [Dij68]. Inter-process messages can be sent. The kernel has an internal memory and stack manager. It supports timesharing, multiprogramming, and priority controls over processes. Interrupts, exception-handling, device-independent I/O, and network communication facilities are included. Frame-level network protocols are available for reliable communication. Disk driving software and a simple file system has also been developed. The version of XINU described in [Com84] implements no protection schemes and does not support multiple users.

With these basic facilities, each CPU can manage its own processes, memory, and I/O requirements. The parallel features of the operating system are implemented at the higher *PASM Operating System (PASMOS)* level. These are the master and server routines for creating and coordinating parallel processes, managing parallel memories, and implementing a distributed file and I/O system. Of course, each of these parallel functions is implemented by appropriate calls to the lower-level (XINU) kernel communication and management routines. For example, to schedule a parallel user process, an operating

system process is begun to communicate the scheduling information to one or more MCs. This operating system process depends on the lower-level XINU process synchronization, reliable communication, and memory management routines to run. When an MC receives a message telling it to start the user process, it does so using system calls implemented by the lower-level kernel functions. Such a scheme implies that processors in the system will communicate using *messages* which, when interpreted, cause *local servers* to be executed to perform the task. In short, XINU is the operating system that runs local to a processor and which supports I/O with other processors. PASMOS is the distributed operating system built upon the facilities provided by individual XINU kernels.

The author has investigated XINU in detail and has determined that major efforts must be made to port it to the PASM hardware. Yet, these efforts are less than those needed to port UNIX to the prototype or to write an operating system "from scratch." A narrative description based on a set of working notes outlines the flaws exposed, proposed solutions, and enhancements made to XINU prior to the time of this thesis writing appears in Appendix A1.5. The notes assume a rather intimate knowledge of the XINU concepts, data structures, and implementation. It is presented in lieu of the modified XINU code itself due to copyright restrictions.

The following subsection summarizes the system calls that are required in an extended XINU kernel suitable for use in the PASM prototype. In a later subsection, the full description of a set of processes that are to comprise the distributed PASMOS operating system is given.

XINU Kernel Requirements

XINU syscalls are organized into the following groups: (1) process management; (2) process coordination; (3) memory management; (4) timer and clock management; (5) device I/O; (6) disk file I/O; (7) inter-process communication; and (8) frame and internet level communication.

Process management includes the following syscalls defined in [Com84].

chprio -- change the priority of a process
 create -- create a new process
 getpid -- return the process id of the currently running process
 getprio -- return the scheduling priority of a given process
 kill -- terminate a process
 resume -- resume a suspended process
 suspend -- suspend a process to keep it from executing

The XINU create() syscall names an entry point in the same object file of the caller where a new process is to begin executing. The caller continues to execute the original process. Variables in the object file are shared among all the processes created during the execution of the object file. There is no capability for creating a child process such that it does not share memory with the parent. Also, there is no provision in XINU to begin execution of a different object file.

Clearly, the existing process creation facilities of XINU are too primitive for use in PASM. A syscall like the UNIX fork() is needed to create an identical copy of a process such that the two do not share data. A shared-data equivalent of fork would also be useful. Finally, a way of naming an independent object file to be executed is needed as in the UNIX exec() syscall.

Much work has already been done to extend the process management in XINU for multi-user operation. Mainly, these extensions protect processes belonging to different users from each other; i.e., disallowing calls to kill(), chprio(), etc. that do not apply to one's own processes.

Process coordination includes the counting semaphore operations.

scount -- return the count associated with a semaphore
 screate -- create a new semaphore
 sdelete -- delete a semaphore
 signal -- signal a semaphore
 signaln -- efficiently signal a semaphore multiple times
 sreset -- reset semaphore count
 wait -- block and wait until semaphore signaled

These are a sufficient set for synchronization among a set of related processes that all know a semaphore's identifier. Some work has already been done to extend these for multi-user operation. Mainly, these extensions protect processes belonging to different tasks from using semaphores not assigned to them. Processes within a task can use semaphores freely because they are all descended from a common ancestor.

One feature that XINU lacks is an asynchronous event signaling mechanism. XINU handles interrupts and asynchronous events only at the operating system level for I/O operations and clock processing. It is often useful for a process to catch and process its own interrupts to be able to define a user-specified recovery from floating point overflow, and the like. On the other hand, UNIX does not provide semaphores to users; it presents only the syscalls for asynchronous event signaling. These are signal(), wait(), and wait3() (not to be confused with the XINU signal() and wait()).

The syscalls associated with memory management in XINU are as follows.

freemem -- return a block of main memory
getmem -- get a block of main memory
getstk -- get a block of main memory for a stack area

While these functions are necessary and sufficient, their current implementation does not set any memory management protection hardware nor make any checks to disallow exhaustive memory allocation. This can lead to deadlock due to multiple outstanding memory allocation requests. There are no provisions for swapping data to disk in the event of deadlock. The inability to detect stack operations overwriting active memory areas is also a problem. Lack of memory management hardware on the prototype prevents a reliable operating system from ever being implemented.

Timer and clock management syscalls include the following:

sleep -- go to sleep for a number of seconds
sleep10 -- go to sleep for a number of tenths of a second

XINU has no provisions for user programs to set and test interval timers. In [Com84], only operating system routines initialize and use the clock to detect expiration of a process' time slice. Work performed by the author allows users to set and test timers explicitly via calls to device I/O routines. This allows execution-time monitoring and evaluation. The periodic asynchronous signaling of a process is not implemented because of the limitations inherent in the process coordination primitives (described earlier).

Device and file I/O syscalls include the following.

close -- device independent close routine
control -- device independent control routine
getc -- device independent character input routine
open -- device independent open routine
putc -- device independent character output routine
read -- device independent input routine
seek -- device independent position seeking routine
write -- device independent output routine

The meanings of device and file are quite different for UNIX and XINU. In XINU, all device I/O functions take a small integer *device descriptor* to indicate for which device I/O service is requested. Each physical device in the system has a different number which the user must embed in programs. A fixed number of device descriptors are dedicated to "file slots." When a disk file is open()-ed, one of these "file slot" device descriptors is used. While this scheme is quite straightforward to implement, it places great demands on users to remember the correspondence between device descriptors and physical devices and to determine which file slots are occupied at any given time so that no conflicts occur. Furthermore, there is no concept of "standard input" and "standard output" being associated with a user's terminal device. This "static assignment" of devices and files to descriptors is a completely unacceptable solution for a multi-user system.

By contrast, UNIX provides dynamically-assigned *file descriptors* when the UNIX open() is called. Both files and physical devices have file names. (Physical device file names begin with "/dev".) The operating system maintains the correspondence between a file descriptor and the device driver routines that control the terminal, disk, or other device being referenced. In addition, the operating system automatically sets up three file descriptors (standard input, output, and error output) that correspond to the user's login session terminal device. Clearly, the UNIX approach is more flexible.

Another flaw in the XINU approach is the use of the syscalls putc() and getc(). Syscalls induce rather high overhead; thus, to require a syscall each time a single character is read or written is rather inefficient. UNIX implements only the read() and write() syscalls; putc() and getc() are user-level library functions that perform buffering to reduce the number of syscalls needed.

The author has performed a number of steps toward adopting the UNIX file descriptor approach in XINU. This has immensely improved the portability of serial programs written for UNIX (to XINU-based systems). I/O operations are a crucial part of all programs; therefore, compatibility in this respect is far more important than in some other set of syscalls (e.g., process coordination).

The set of device drivers associated with a kernel depends on the devices found on a given CPU-memory-I/O system. Device drivers do not necessarily control real peripheral devices. For example a *RAM disk* driver is used in the PEs and MCs to associate calls to `open()`, `read()`, `write()`, etc. with data files stored in their own RAM memories. Programs manipulate the files through these syscalls as if the files were actually stored on physical disks associated with the CPU. The RAM disk driver in a PE does not manipulate a disk drive at all: it either performs memory block move operations locally or sends messages to the MC that eventually cause an MSU to load data in the PE's memory.

In the prototype PEs, device drivers for the serial diagnostic port, the local clock, the parallel port interface to the network, the MC-PE I/O port, the Condition Code Register and Logic, the performance timers, and the RAM disk will be included. Prototype MCs will have device drivers for their serial diagnostic port, local clock, parallel port communication channels, MC-PE I/O port, Data Conditional Mask Register, Immediate Broadcast Register, Enable Signal Register, performance timers, FBU interface, RAM disk, and so on. Control Storage and the MSUs will have device drivers for their serial diagnostic port, local clock, parallel port communication channels, performance timers, Disk Controller interface, and Disk Access Switch registers. The SCU has device drivers for its serial diagnostic port, local clock, parallel port communication channels, performance timers, RAM disk, optional peripherals such as a system time-of-day clock, printers, tape devices, and displays, Ethernet interface, console port, and so on. Each prototype component will, in general, have differing numbers of each type of I/O device which will be installed at different physical addresses in the address space. Much work has already been done by the author to develop automatic XINU boot-up system configuration routines that can detect which types of devices are connected to the system by examining the plausible physical addresses at which they might appear.

Disk file I/O syscalls were discussed in the previous paragraphs. Both those kernels that have a RAM disk (PEs, MCs) and those that have a physical disk (Control Storage, MSUs) will use the same file I/O syscalls to access a file. For the RAM disk, an open() generates a request handled externally that eventually results in the named file being made resident. For the physical disk, an open() is handled locally. It causes physical disk accesses to find the file in the file system structure, and associates a name with a file descriptor. A close() on a RAM disk driver causes the file to be unloaded under the control of an external processor. Close() for a physical disk breaks a name-file descriptor link and migrates any modified disk blocks back onto the disk. A read() or write() for a RAM disk simply moves data from buffer to buffer within memory. For a physical disk driver, read() and write() generally cause a number of disk accesses to be performed. When reading, the system determines where the next disk block is located (they are usually stored non-contiguously), extracts the block, and copies it to the user's buffer. When writing, free disk blocks must be identified and linked into the existing file structure.

Inter-process communication syscalls in XINU are as follows.

```
pcount -- return the number of messages currently waiting on a port  
pcreate -- create a new port  
pdelete -- delete a port  
preceive -- get a message from a port  
preset -- reset a port  
psend -- send a message to a port  
receive -- block until a (one-word) message is received  
recvclr -- return a (one-word) message if one is present  
send -- send a (one-word) message to a process if it can accept one  
sendf -- force sending a (one-word) message to a process
```

XINU has two groups of inter-process message facilities: one that uses "ports" that buffer multiple messages of any length and one that uses a one-word "mailbox" associated with each process. The first approach is certainly more flexible, but has higher overhead. The second approach is not very useful for user-oriented programs and exists primarily as a rudimentary signaling device for low-level operating system processes. It is the author's intention to embed the receive(), recvclr(), send(), and sendf() calls within XINU to disallow users from accessing them. Users will make syscalls only to the "port" functions. UNIX implements a port mechanism that includes both pipes and inter-processor ports (to be described).

Finally, XINU dedicates a fixed number of device descriptors for inter-processor communications. When one of the device-independent I/O functions such as `read()` or `write()` is used and the device descriptor specifies a "network" connection, a *packet* consisting of a *packet header* and a *message* is formed. The packet header indicates the process id that is to receive the message. The packet is then prefixed with an *internet header* specifying the destination machine to form a *datagram*. Datagrams are prefixed by a *frame header* that contains control information such as frame sequence numbers (to detect lost frames) and if necessary, an intermediate machine name (if the source and destination machines are not directly connected). Finally, the frame is embedded in a *block* which is actually transmitted over the physical network hardware (indicated by the device descriptor) to the destination or to an intermediate machine.

The XINU syscalls for frame-level network communication are given below.

`freceive` -- receive the next frame that arrives from a network
`fsend` -- enqueue a message for transmission to another machine
`finit` -- initialize frame-level network input and output processing

The organization of the XINU networking leaves much to be desired. It assumes a unidirectional ring network connects the processors. This allows the issue of routing to be ignored because there is only one "route." Routing tables have to be substituted in the PASM point-to-point communication scheme. Also, the frame-level routines do not address the need to construct and extract messages from packets and packets from datagrams. UNIX has a number of system calls dedicated to this type of network communication that are more reliable than the ones provided by XINU. They use names in the file system space as rendezvous points for incoming messages. UNIX syscalls allow a user to choose one of a number of protocols and address formats. For the PASM prototype, only the virtual circuit protocol needs to be implemented.

In the next subsection, the basic operations provided by a XINU kernel (extended in the ways described above) are used to propose an organization for PASMOS. The organization assumes the hardware and configuration of the PASM prototype with a network connection between a VAX/UNIX Engineering Computer Network host computer and the PASM SCU.

PASMOS Processes

PASM users will begin by establishing a terminal session on the ECN host computer to which PASM is attached. Because the PASM prototype does not run UNIX, a virtual terminal cannot be provided on the SCU itself (e.g., via UNIX rlogin or rsh). Instead, a special multiplexed communication protocol is established between the ECN host and the SCU. A PASM command processor program *psh* (PASM shell) on the ECN host is executed by each PASM user. Upon entry, *psh* sends a message tagged with the user's login id to the PASM SCU requesting that a PASM session be established. If the SCU accepts the user (possibly requiring a password or other identifying information), a session is established and PASM returns a session number to the shell. On the PASM SCU, a conventional shell (e.g., sh or csh) for the user is started. Hereafter, the shell on ECN that the user directly interacts with is referred to as *psh* while the "remote" shell on the PASM SCU will be known as *sh*.

Commands typed to *psh* are executed locally as in a normal shell except if they begin with the special escape character "@"." The escape character prevents local interpretation and instead encapsulates the command string in a message tagged with the session number and passes it to the SCU. A process that continually runs on the SCU accepts messages of this type from all user's sessions and passes them to the *sh* associated with the user's session. This protocol simulates independent remote shells for multiple users on PASM. Certain built-in commands to *psh* specify the movement of data or program files between PASM and the host. Responses by PASM back to the user are also encapsulated and returned to a server process on the host to be distributed to the appropriate *psh* session.

Psh command syntax is similar to that of normal shells. The first word on a command line gives the name of a program to be run. I/O redirection and pipes are allowed. Background processes may be started. Job control is to be supported.

To execute a command, *sh* must decide whether it is an object file or a command script, execute a *fork()* to obtain a copy of *sh* and *exec()* the child into the desired command. Object files have headers of a prescribed form and can be readily identified as serial or SIMD programs. (MIMD programs are always started as SIMD programs that spawn the MIMD processes) Serial

programs are executed by the SCU independently, although the SCU may call upon the MCs, MMS, or other processors for information. An example is a command that requests a file directory listing. The program that executes on the SCU gathers the information from Control Storage and the MSUs, formats it, and returns it to the user. SIMD programs require *sh* to pass the command line to the *Parallel Computation Unit Scheduler*, a process that runs continuously on the SCU and monitors all activities of the MCs and PEs. This scheduler determines the partition based on system load, availability of MCs, the needs of the SIMD process (number of VPEs specifier in the header), the "health" of the system in terms of memory capacity and correctly-functioning MC-groups, and so on. By "continuously running" it is meant that the process is created at PASM boot-up time and does not terminate. However, it likely spends most of its time "blocked" and not actually consuming processor cycles. The Parallel Computation Unit Scheduler is not to be confused with the XINU scheduler on the SCU; a XINU scheduler is part of the XINU kernel and determines when the Parallel Computation Unit Scheduler runs, when a serial process runs, when the user *sh*'s run, and so on. The Parallel Computation Unit Scheduler becomes active when *events* occur. An event is a parallel task initiation, parallel task completion (normal or otherwise), a change in parallel task priority, a change in the health of the system (affecting scheduling discipline), or the end of a *rescheduling interval*. A rescheduling interval would be a periodic event that would allow the Parallel Computation Unit Scheduler to re-evaluate its priorities, swap tasks off the system, and so on.

When a parallel task is scheduled, each MC in the scheduled partition is informed. The *VMC Scheduler* is a continuously-running MC process that communicates with the SCU, MMS, other MCs (via the SCU), and its PEs. It implements the co-scheduling protocol necessary when multiple LMCs each simulate multiple VMCs. MCs always run with interrupts enabled so that the VMC Scheduler can respond to messages in a timely manner.

PEs contain no continuously-running processes: all communication with it is synchronous in SIMD mode and interrupt-driven in MIMD mode. The XINU kernel in a PE is active only during syscalls or when context-switching between multiple MIMD processes.

All of the MMS processors, MSUs, and Control Storage processors have a single continuously-running process that is normally blocked but which becomes active when initiated by an incoming request. Consider for example the Command Distribution Processor. When an incoming request is received, the "request interpreting process" becomes active. If it is determined that all of the MSUs are to be started on some operation, a child process would be spawned to send out the requests and to handle the return acknowledgements. (Alternatively, N/Q separate children could be spawned, one for each MSU). The child would block while waiting for acknowledgements. When the acknowledgements have all been received, the child sends an acknowledgement back to the Memory Scheduling Processor and dies. Similar protocols would be adopted for the other processors in this class.

Left unspecified in this description is the format of messages passed among the processors of PASM. Also unspecified are schemes for operating system error recovery in the event of lost messages, PASMOS crashes in one or more processors, disk errors, and the like. A robust distributed system is exceeding difficult to design and implement: consider the thousands of man-years spent writing and debugging code for the Arpanet for example.

2.5.7 Summary

This section developed schemes to handle the interactions of the PASM Memory Management System, secondary storage devices, and other subsystems involved in the transfer of program and data files between primary and secondary memory. Conventions for distributing data and program files among multiple secondary storage devices were discussed. A distributed approach was taken in order to provide much computational power at low cost. This approach also removes the burden of memory management from the SCU and provides for file input, output, and reformatting without interruption of the PEs. The distributed approach is likely to be suitable for other parallel processing systems requiring high throughput as well. In the last section, the XINU and PASMOS operating system levels were described and the division of labor among the various PASM processors was outlined.

CHAPTER 6

SUMMARY

This part of the thesis has considered how PASM is to appear to the system and application programmer. The language and operating system are important aspects of the PASM development because they are the primary interface between the designers of the hardware and the researchers and students who will finally use the system. Because PASM is a research machine the languages and operating system must provide several levels of support. For the applications programmer who is simply interested in fast execution, standard languages and operating system environments will be provided. However, the systems programmer exploring new schemes for scheduling parallel processes will require a very different set of tools and capabilities. For this reason, a variety of issues and solutions in the language and operating system domains are being studied and implemented. Ongoing research by others includes the problems of scheduling processes onto processors in MIMD mode and the capabilities of the interface that the user sees during a PASM "terminal session."

PART III

PARALLEL IMAGE PROCESSING ALGORITHM STUDIES

CHAPTER 1

INTRODUCTION

An important part of the PASM development has been the study of image processing applications and their execution on PASM. Partly, these algorithm studies were done to determine their implicit parallelism and therefore, how efficiently they would be executed on an SIMD/MIMD system such as PASM. Both a complexity analysis and simulation can be used to satisfy this need. Another function of the algorithm studies has been to determine what architectural features in PASM seem to lacking. This has driven the design of PASM and has provided important feedback on the prototype implementation.

In this part of the thesis, program development aids, algorithms, and simulation results are described. Chapter 3.2 discusses the attributes of the PASM simulator that was used to perform the studies. Performance measures for SIMD/MIMD processing are introduced in Chapter 3.3. Five different SIMD and MIMD algorithms are discussed in Chapters 3.4 through 3.8. Each algorithm is presented by a functional description, followed by a discussion of its execution characteristics as determined through simulation. Some of the material in Part III has been previously published in [KuS81, KuS82, KuS84, KuS85, KuF85, KuS86b].

CHAPTER 2

DEVELOPMENT AIDS

The algorithm simulations were carried out using a specialized simulation facility developed in-house by the author. Early simulation work [SiK80, SiK81, SiK82] using existing simulation programs was often less than completely successful because such programs either limited the number of processors that could be simulated, the size of the data sets that could be processed, or the timing information that could be obtained. Another limitation of existing simulation systems was that to declare and initialize a system of N processors, N individual commands often had to be given. This is inconvenient when N is large. It was obvious that the simulators that were evaluated were built with small multiprocessing systems in mind rather than the massive synchronized parallelism in systems such as PASM. For this reason, a more efficient generalized simulator, *psst* (Purdue SIMD/MIMD Simulation and Timing) was developed to allow the simulation of serial and parallel architectures in general and 68000-family processors acting as SIMD/MIMD machines in particular. Details of the simulator's functions and features are given in its manual found in Appendix A1.6. PSST code is given in Appendices A2.2 through A2.4.

Conceptually, *psst* is the name of a simulation program that is link-loaded from two sources. One part (called PSST) contains the user interface, "glue" routines, implementation of communication structures such as buses and ports, and a scheduler that provides the illusion of a number of processors executing concurrently. The other part is taken from a number of "device libraries" that describe the specific characteristics of the hardware to be simulated. To simulate two MC68000 CPUs connected by a port, the copy of the "MC68000" simulation is extracted from the device library and link-loaded with the PSST routines to form an executable *psst* program. Details of the library format,

contents of the libraries, "glue" routines, and the linking process are given in the manual.

The user specifies the interconnection *topology* of the system and provides the instruction and data streams for the simulation. During a simulation, any device internal register or memory may be examined. The simulation can also be single-stepped through each instruction to aid the debugging of algorithms. Breakpoints are also available. The simulator provides output of the processed data streams as well as the number of internal cycles the simulation required. The number of cycles is related to the time in seconds by the clock frequency.

Pssst is a general MIMD simulator, yet most of the simulation work performed by the author has been with SIMD algorithms. SIMD mode is simulated by defining components that force synchronization. For example, there is a device used in SIMD simulations that requests a PE instruction from the FBU device only when all PEs have received and executed the previous instruction. Operating system mapping of multiple VPEs to LPEs is not done: there is an assumed one-to-one mapping of VPEs to LPEs.

MIMD operation is simulated although with no interrupts. Therefore, MIMD algorithms must poll their I/O ports for data rather than depend on being interrupted when data arrives asynchronously. The 68000-family components in the library handle exceptions and perform exception processing (which interrupts also cause); however, there is no current way to pass interrupts between components.

An MIMD mode program to be simulated is obtained by changing the SIMD algorithm so that no masking operations are performed and the program control operations are done by the PEs themselves. In addition, at each point in the code that a PE would retrieve an item transferred to it through the interconnection network, a "wait-loop" was added. The loop is executed until the data arrives allowing synchronization of PEs based on message passing. In a practical implementation, an interrupt would be generated at the destination PE when an incoming data item arrived. The simulator does not take into account instruction execution timings that are data dependent (e.g., multiplication and division). This implies that the effects of PEs being idled in SIMD mode while waiting for other PEs to finish the current instruction are not taken into account. The simulator also fails to model the effects of conflicts

internal to the interconnection network. The interconnection network transfer time includes that for a network setting to be established, the time for the data item to be placed in the network input port, the time for the data item to traverse the network, and the time to retrieve the data item at the network output ports. The times for the data to be placed in the input and to traverse the network are identical for SIMD and MIMD modes; however, the other times are longer for MIMD mode. A most conservative assumption is to equate the MIMD transfer time with that of the SIMD transfer time. All of these factors taken together imply that the MIMD simulation results underestimate the actual algorithm execution time.

The interconnection network is treated as one functional unit, i.e., switches and links internal to the network are not simulated. Therefore, interconnection network "conflicts" cannot be detected. However, this is not seen as a disadvantage since all types of networks can be simulated efficiently and without modification to the simulator program. It is the user's responsibility to assume a certain interconnection network type and then to write algorithms that use only the inter-PE transfers allowed by that network. For example, if the multistage cube network is assumed, a data transfer from PE i to PE $i+2 \pmod{N}$ can be performed in one inter-PE data transfer. On the other hand, if the Illiac network is assumed (which can only do transfers of $i+1$ and $i\pm 8$), the same data movement would take two steps (inter-PE transfers). Both steps should be written explicitly in the assembly language algorithm so that the timing can be calculated correctly. The time to traverse a network is a parameter that can be set to reflect different network assumptions.

In order to discuss and compare the performance measures, several example SIMD and MIMD image processing algorithms are examined. These are: (1) parallel image smoothing; (2) parallel global image histogramming; (3) edge-guided thresholding/contour tracing; (4) vector-to-raster conversion; and (5) raster-to-vector conversion (thinning and vectorization).

The image smoothing, histogramming, edge-guided thresholding, and thinning algorithms have been written in the parallel MC68000 assembly language (for SIMD processing) and simulated. Contour tracing has been studied and is an extension of the work in [TuA83] but is not simulated. The vector-to-raster algorithm and the vectorization algorithm of the raster-to-vector conversion

task are written in standard C; however, they attempt to simulate these algorithms' execution on a parallel processor.

CHAPTER 3

PERFORMANCE MEASURES

3.3.1 Introduction

In this chapter, measures for evaluating algorithm performance for SIMD and MIMD machines are examined. In general, the performance is a function of the machine size, problem size, the mode of processing (SIMD or MIMD), the type of interconnection network, and the degree to which the system hardware allows operations to be overlapped or pipelined. The notions and interrelationships of measures such as algorithm execution time, computation time, overhead time, speedup, efficiency, and processor utilization are examined for several SIMD and MIMD machine models. The PASM parallel processing system is used as an example SIMD/MIMD machine.

3.3.2 Performance Measures

The *execution time* $T_N(P,M)$ is defined to be the time required to perform an algorithm for a problem of size P on an N -PE machine [SiS82b] using mode of parallelism M ($M \in \{\text{SIMD/D, MIMD}\}$). The notation "SIMD/D" refers to SIMD mode computation with degree of decoupling D between the MC and PEs.

$T_N(P,M)$ can be thought of as the sum of two components: $c_N(P,M)$, the "computation" time spent performing calculations directly relevant to the problem (including loop control operations), and $o_N(P,M)$, the "overhead" time spent in managing the parallelism [SiS82b] or accommodating an inequitable distribution of resources, data, or computation among the PEs. The overhead is primarily due to:

1. interconnection network operations needed to transfer data from one PE to another (both SIMD and MIMD modes);
2. masking operations used to control which PEs are enabled/disabled during the course of the algorithm execution (SIMD mode);
3. process synchronization operations (MIMD mode); and

$\alpha_1(P,M)$ is assumed to be zero since there are no inter-PE transfers nor is there a need to disable or synchronize the single PE.

In general, the computation time decreases as N increases since as more PEs are applied to a problem, each PE performs fewer operations. However, the computation time also depends on the mode of parallelism and if SIMD, the degree of decoupling between the MC and PEs. Consider the following program that contains both arithmetic calculations (e.g., add, load/store) and program flow operations (e.g., initialization, test an index variable, branch). Expressed in a serial high-level language:

```

for i := 0 to N-1
    for j := 0 to 4
        C[i][j] := A[i][j] + B[i][j];
    
```

For a serial machine, $5N$ addition steps and the program flow overhead of two nested indices is required. On an SIMD or MIMD machine, A, B, and C can each be thought of as arrays of five elements in each of N PEs: PE i holds $A[i][j]$, $B[i][j]$, and $C[i][j]$, $0 \leq j \leq 4$. Thus the SIMD/MIMD algorithm could be expressed as:

```

for j := 0 to 4
    C[j] := A[j] + B[j];
    
```

which requires five addition steps for an N -PE machine since each PE performs the loading, adding, and storing operations simultaneously.

For this example, there is also a reduced program flow overhead for the parallel algorithm as compared to the serial one because a loop is *unwound* across the extent of the parallelism. In the parallel case, there is a single index variable to be handled and the number of increment/test operations performed is five. In the serial case, the inner loop accounts for $5N$ increment/test operations and the outer loop for an additional N , making a total of $6N$. One might conclude that this demonstrates an advantage of parallel processing: the

number of operations decreases by more than a factor of N when N PEs are used. However, notice that the serial algorithm *could* have been written with a single index variable ranging from 0 to $5N-1$ ($5N$ operations). For the purposes of this discussion, it is assumed that the serial algorithm has been adjusted so that both the serial and parallel versions have the same degree of loop nesting.

Expressed in a pseudo-assembly language for an SIMD machine, the task is:

```
c_move #0, j
L1: c_cmp j, #5
      c_bge L2
      p_move A[j], R0
      p_move B[j], R1
      p_add  R0, R1
      p_move R1, C[j]
      c_add  #1, j
      c_bra  L1
```

L2:

The "c_" prefix designates instructions executed by the MC while the "p_" designates instructions broadcast to and executed by the PEs. The entire program is held in the MC memory. The MC fetches each instruction, determines whether it is to be executed in the MC or PEs (the assembler tags each instruction accordingly), executes the control flow (c_) instructions itself, and broadcasts the processing (p_) instructions to the PEs. The list of instructions broadcast to and received by the PEs as a result of the MC executing the program above are:

```

move A[0], R0
move B[0], R1
add R0, R1
move R1, C[0]
move A[1], R0
move B[1], R1
add R0, R1
move R1, C[1]

.
.
.

move R1, C[4]

```

Note that the PE receives and executes only the arithmetic calculations; the MC executes the program flow operations itself.

If the complete parallel pseudo-assembly language program given above were to be executed in MIMD mode, it would be contained in each PE's memory (minus the prefix tags). This illustrates an advantage of SIMD computation: only one copy of the program need be stored in the MC. Since the PEs are responsible for performing their own loop counting and branching in MIMD mode, the execution time of the task in SIMD mode is potentially less than in MIMD mode if the actions of the MC and PEs can be overlapped. The decoupling (D) of the MC and PEs represents the amount of overlap that the machine allows. In general, as more overlap is available, the computation time decreases.

Several different hardware configurations each representing a different case of decoupling were presented in [KuS82]. In the "non-decoupled" case, either the MC operates or the PEs operate, but not both at the same time. This situation represents no improvement over the MIMD mode computation time. For the "fully-decoupled" case, a FIFO queue having infinite length is assumed to exist between the MC and PEs. The MC enqueues PE instructions when it encounters them in the instruction stream and the PEs dequeue instructions as they require them. An instruction is dequeued when all enabled PEs request one [KuS82]. Earlier studies have determined that the FIFO queue needs to hold only the number of PE instruction words that would be required to

prevent the PEs from "starving" during the longest sequences of MC instructions [KuS82] to achieve the maximum decoupling. Cases of "intermediate-decoupling" occur when the queue is of insufficient length to keep the PEs from being starved of instructions. The size actually required for maximum decoupling varies from problem to problem; however, for the earlier algorithm simulations performed, a queue capable of holding five to ten average MC68010 instructions is sufficient. Because the instructions are actually enqueued/dequeued in units of 16-bit words and each instruction is from one to five words, the queue length is typically expressed in units of words. Ten average MC68010 instructions are equivalent to about 30 words.

The overhead time ($\alpha_N(P,M)$) due to interconnection network transfers depends on the mode of parallelism being employed. In SIMD mode, PEs are synchronized; therefore, interconnection network operations are performed such that all enabled PEs participate in the transfer of data simultaneously. The degree of decoupling does not affect this overhead. However, the actual transmission time of data through the network may be overlapped with the PE computations. The programmer/compiler arranges an SIMD transfer such that the data can be transmitted in one or more non-conflicting "passes" through the network. A network "conflict" occurs when two or more data items need the same internal link simultaneously. In MIMD mode, PEs run asynchronously; thus, messages sent from a source PE to a destination PE through the interconnection network may encounter network conflicts. When a conflict occurs, one message is allowed to continue; other messages are forced to wait for the availability of the network link. Arbitration and conflict resolution in the network may result in higher overhead time for MIMD tasks.

Another network-influenced overhead time encountered in MIMD processing occurs when a message arrives at a destination PE. The arrival results in an interrupt being generated at that PE causing it to read and buffer the incoming message. No interrupts need be generated in SIMD mode; the instructions to read the network output are written explicitly in the SIMD program. Furthermore, the source of a message is known to the destination PE in SIMD mode implicitly because the order in which the transfers are executed is the same order in which the data is received. This is not the case in MIMD mode: messages may arrive in any order (due to the independence of the PEs)

and therefore must contain the number of the source PE. An additional overhead time results in MIMD mode because the source PE number must be read and processed explicitly.

The overhead due to SIMD masking operations involves the time spent by the MC in determining which PEs should be enabled/disabled and in constructing the "mask." If there is decoupling between the MC and PEs, the effects of this overhead may be reduced since it may be overlapped with PE computations. Masking operations may be necessary if there is an inequitable distribution of data (i.e., PEs finished with processing must be disabled) or if there is a difference in the computations required (for example, PEs having image edges perform different computations).

An analogous overhead in MIMD mode is that due to synchronization operations. In contrast to the SIMD masking operations, synchronization operations are used to enforce an ordering among the separate instruction streams. No such ordering is required for the single SIMD or serial instruction stream. Usually the ordering is used to indicate the availability of shared data. For example, one process will signal to others that it has reached a certain milestone in the program indicating that shared data has been processed. If another process has already reached the synchronization point, it would have been "busy-waiting" for the signal to arrive. On the other hand, if a process receives the signal before reaching the synchronization point, it will not have to busy-wait since the shared data is already available. The only overhead incurred by such a process would be that for testing whether the synchronization signal was received (which it was). Synchronization operations are actually performed by all participating PEs; however, some PEs' operations may not contribute to the overhead time. Unfortunately, the decision to add or not to add the synchronization instruction time to the overhead time depends on what other operations are being performed in other PEs simultaneously. In general, one can identify a *critical execution path*, the longest path of arithmetic computations. Only the synchronization instructions in this critical execution path contribute to the overhead time.

SIMD speedup is a common measure for expressing how well an algorithm performs on an SIMD system having N PEs (and a MC) as compared to one having a single PE (and a MC). Typically, it has been expressed as the ratio:

$$S_N(P,M) = \frac{T_1(P,M)}{T_N(P,M)}$$

which has values in the range $0 \leq S_N(P,M) \leq N$ [SiS82b]. For this measure, the values of P and M (which includes D) are assumed to be the same for S_N , T_1 , and T_N . The speedup is *ideal* if $S_N(P,M) = N$ and implies that $\alpha_N(P,M) = 0$. *MIMD speedup* has the same formula but compares an MIMD machine with N PEs to one having a single PE.

The *absolute SIMD speedup* compares the SIMD parallel algorithm to the equivalent serial one. "Serial" in this context refers to a single PE acting on its own without a MC. $T_{\text{serial}}(P)$ is equivalent to $T_1(P,\text{SIMD/non-decoupled})$. This type of speedup is expressed by the ratio:

$$AS_N(P,\text{SIMD}/D) = \frac{T_{\text{serial}}(P)}{T_N(P,\text{SIMD}/D)}$$

which has values in the range $0 \leq AS_N(P,\text{SIMD}/D) \leq N$ for SIMD machines with no decoupling (consistent with the SIMD speedup measure given above) and $0 \leq AS_N(P,\text{SIMD}/D) \leq 2N$ for SIMD machines with full decoupling.

For an SIMD machine, it may seem counter-intuitive to be able to achieve a $2N$ speedup over the serial algorithm using only $N+1$ processors (configured as a MC and N PEs). The seemingly "better-than-ideal" speedup can be explained by considering Figure 3.3.1. Figure 3.3.1a is a model for a 1-MC 1-PE SIMD machine that has the potential for full decoupling. Thus it may perform an SIMD algorithm in which MC and PE instructions can be exactly overlapped twice as fast as the 1-PE (serial) model of Figure 3.3.1b. Figure 3.3.1c is seen to be up to N times as fast as the machine of Figure 3.3.1a; thus, it is up to $2N$ times as fast as the model of Figure 3.3.1b. However, by the definition of SIMD processing, the model of Figure 3.3.1c is equivalent to that of Figure 3.3.1d since each of the MCs of Figure 3.3.1c would perform the same operations at the same time. Thus for the "price" of $N+1$ processors configured as an SIMD machine, up to a $2N$ speedup may be obtained over a serial processor.

The *efficiency* $E_N(P,M)$ of an N-PE algorithm is defined to be

$$E_N(P,M) = S_N(P,M)/N$$

and has values in the range $0 \leq E_N(P,M) \leq 1$ [SiS82b]. Intuitively, efficiency

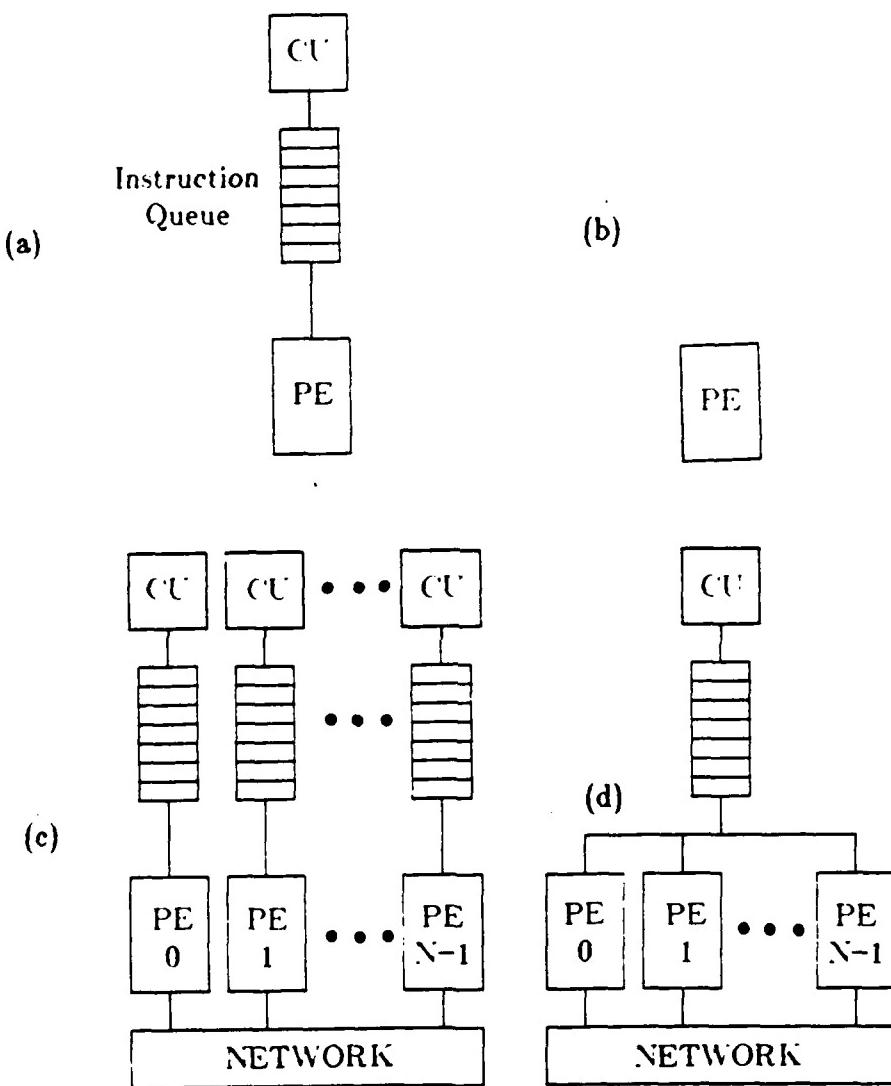


Figure 3.3.1. (a) 1-CU and 1-PE SIMD decoupling model. (b) Serial model.
 (c) N-CU and N-PE SIMD decoupling model. (d) 1-CU and N-PE SIMD decoupling model.

measures the ratio of the speedup ($S_N(P,M)$) compared to the ideal speedup (N). The *absolute SIMD efficiency* is defined as:

$$AE_N(P,SIMD/D) = AS_N(P,SIMD/D)/2N.$$

This measure includes the effects of the control operations and indicates how well all of the processors (MC and PEs) are being used.

The *utilization* $U_N(P,M)$ measures the fraction of the time during which the PEs are actually executing algorithm computations. In SIMD mode, PEs are idle when they are disabled, waiting for an instruction from the queue, or waiting for other PEs to finish the current instruction. In MIMD mode, PEs are idle when they are "busy-waiting." As suggested in [SiS82b], utilization can be calculated by counting the number of PEs active for each computation step. The "step" must be a fixed-length time such as a machine or memory cycle so that the SIMD and MIMD utilizations can be compared. Assuming the time $T_N(P,M)$ is composed of X steps (clock cycles) and the count of the active PEs during step x , $0 \leq x < X$ is given by p_x , the utilization can be defined as:

$$U_N(P,M) = \sum_{x=0}^{X-1} p_x / NX.$$

Finally, the *absolute SIMD utilization* can be defined as:

$$AU_N(P,M) = \sum_{x=0}^{X-1} p_x / (N+1)X$$

where the count of the active processors includes the MC. Utilization measures have values varying between zero and one, with unity being the ideal utilization. Situations during which the MC may be idled are given below.

1. The MC may be waiting for the PEs to empty the instruction queue and perform some test on their local data. The result of this test may be needed by the MC to determine the global state of the PEs, for example, "if any" PE has identified an object of interest in its part of an image.
2. The MC may have filled the instruction queue to capacity and is waiting for the PEs to execute instructions to make some space in the queue.

3. The MC may have finished all of its own instructions and placed the last PE instruction in the queue; however, the execution of the program is not complete until the PEs have completely drained the queue.

An ideal absolute SIMD utilization is difficult to achieve in practice due to these effects.

3.3.3 Summary

Performance measures such as execution time, speedup, and utilization can be used to help understand the interactions of an algorithm with a particular parallel processing system. The earlier work on SIMD performance measures was based on a complexity analysis of algorithms and included the effects of arithmetic operations, masking operations, and network operations. Simulation studies and further analysis led to enhanced performance measures for SIMD mode and an extension of the measures for MIMD mode. This chapter demonstrated that the components of and relationships between these measures are often complex. Clearly, performance measures which deal with other system components such as I/O will be needed to further aid programmers in analyzing how an algorithm of interest will perform.

CHAPTER 4

SMOOTHING ALGORITHM

Two versions of an SIMD image smoothing algorithm for a 16-PE system were simulated and described in [SiK81]. Based on those results, one algorithm appeared to give better performance for realistically-sized images. Further studies were done for machine varying machine sizes and for MIMD mode. It should be noted that these algorithms demonstrate the simulation characteristics rather than exemplify a highly efficient or novel method of performing smoothing operations. The algorithms may be compared for the number of instructions in the static code, the number of instructions performed during execution, and the time required to execute the algorithm.

The algorithm given here "smooths" a gray level input image and is based on one described in [SiS81b]. The algorithm has I as an input image and O as an output image; both I and O contain M -by- M pixels, $P=M^2$. Each point of I is a value representing one of G possible gray levels. Each point in the smoothed output image, $O(i,j)$, is the average of the gray levels of $I(i,j)$ and its eight nearest neighbors, $I(i-1,j-1)$, $I(i-1,j)$, $I(i-1,j+1)$, $I(i,j-1)$, $I(i,j+1)$, $I(i+1,j-1)$, $I(i+1,j)$, and $I(i+1,j+1)$, where $1 \leq i, j < M-1$. Pixels on the edges of I are not smoothed; they are simply copied to O .

Consider how this could be implemented on an SIMD machine with N PEs, logically arranged as an array of \sqrt{N} -by- \sqrt{N} PEs. Each PE stores an M/\sqrt{N} -by- M/\sqrt{N} *subimage* block of the original image I . Specifically, PE 0 stores the pixels in columns 0 to $(M/\sqrt{N})-1$ of rows 0 to $(M/\sqrt{N})-1$, PE 1 stores the pixels in columns M/\sqrt{N} to $(2M/\sqrt{N})-1$ of rows 0 to $(M/\sqrt{N})-1$, and so on. Each PE smooths its own subimage, with all PEs doing this simultaneously. At the edges of each subimage, data must be transmitted between PEs in order to calculate the smoothed value. The necessary data transfers are

shown for PE J in Figure 3.4.1. Transfers between different PEs can occur simultaneously; e.g., when PE J-1 sends its upper right corner pixel to PE J, PE J can send its upper right corner pixel to PE J+1, PE J+1 can send its upper right corner pixel to PE J+2, etc. Cube-type networks can perform these transfers in a single step.

The first smoothing algorithm utilizes a very straightforward approach. The PE memory configurations at various stages of execution of the smoothing algorithms are shown in Figures 3.4.2a through 3.4.2h. Initially, it is assumed that each PE has a subimage in its local memory (Figure 3.4.2a). The edge points of each PE's subimage are transferred through the interconnection network to adjacent processors and stored in a local "work" area (Figure 3.4.2b). The interconnection network is set to perform " $+2 \bmod 4$ " shifts of data. For this reason, image points "wrap around" (e.g., when the top row of pixels in PE 0 is transferred it becomes the bottom row of pixels in PE 2). Each PE then copies its original subimage into the "work" area (Figure 3.4.2c). Then, processors having points on the edge of the total image "fix up" the edge data in the work area by forming a border consisting of adjacent original image data (Figure 3.4.2d). Finally, the windowing operations (a window size of nine pixels is used) are performed on the data in the work area and the results are placed back in the original subimage area (Figure 3.4.2e).

The second algorithm is an improvement on the first, although for small image sizes it performs somewhat worse than the first version. Again, it is assumed that each PE has a subimage in its local memory (Figure 3.4.2a). The edge points of each PE's subimage are transferred through the interconnection network to adjacent processors and stored in a local "work" area (Figure 3.4.2b). Each PE then copies a two-pixel-deep border of its original subimage into the "work" area. Note that there is a savings in the number of pixels copied only when the subimage size is greater than 4 by 4 elements. The two-pixel-deep border allows the outer "ring" of smoothed pixels to be calculated, but reduces the copying of image data (Figure 3.4.2f). Then, processors having points on the edge of the total image "fix up" the edge data in the work area by forming a border consisting of adjacent original image data (Figure 3.4.2d). Finally, the windowing operations are performed on the "ring of data" in the work area (Figure 3.4.2g), and the original image area (Figure 3.4.2h). The

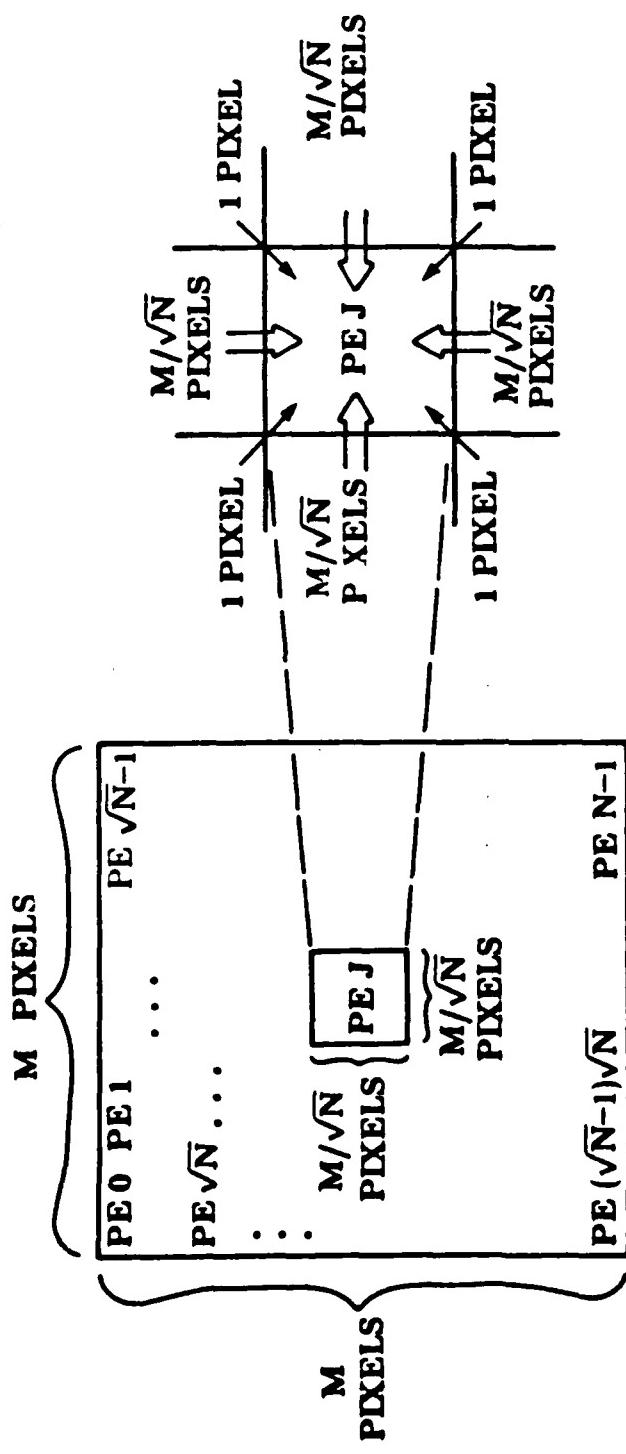


Figure 3.4.1. Data transfers required in PE J for the "checkerboard" data allocation.

| PE0 | PE1 | | |
|-----|-----|-----|-----|
| 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 |

| PE2 | PE3 | | |
|-----|-----|-----|-----|
| 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 |

Figure 3.4.2a. Original subimage arrays (4-by-4).

| PE0 | PE1 | | | | |
|-----|-----|-----|-----|-----|-----|
| 000 | 000 | 000 | 000 | 000 | 000 |
| 255 | | | | 255 | |
| 255 | | | | 255 | |
| 255 | | | | 255 | |
| 255 | | | | 255 | |
| 000 | 000 | 000 | 000 | 000 | 000 |

| PE2 | PE3 | | | | |
|-----|-----|-----|-----|-----|-----|
| 255 | 255 | 255 | 255 | 255 | 255 |
| 000 | | | | 000 | |
| 000 | | | | 000 | |
| 000 | | | | 000 | |
| 000 | | | | 000 | |
| 255 | 255 | 255 | 255 | 255 | 255 |

Figure 3.4.2b. Workspace arrays (6-by-6) after interprocessor data transfers.

| PE0 | | | | | |
|-----|-----|-----|-----|-----|-----|
| 000 | 000 | 000 | 000 | 000 | 000 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 000 | 000 | 000 | 000 | 000 | 000 |

| PE1 | | | | | |
|-----|-----|-----|-----|-----|-----|
| 000 | 000 | 000 | 000 | 000 | 000 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 000 | 000 | 000 | 000 | 000 | 000 |

| PE2 | | | | | |
|-----|-----|-----|-----|-----|-----|
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 000 | 000 | 000 | 000 | 000 | 000 |

| PE3 | | | | | |
|-----|-----|-----|-----|-----|-----|
| 255 | 255 | 255 | 255 | 255 | 255 |
| 000 | 000 | 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 | 000 | 000 |

Figure 3.4.2c. Workspace arrays (6-by-6) after copy of original subimage data.

| PE0 | | | | | |
|-----|-----|-----|-----|-----|-----|
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 000 | 000 | 000 | 000 | 000 | 000 |

| PE1 | | | | | |
|-----|-----|-----|-----|-----|-----|
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 |
| 000 | 000 | 000 | 000 | 000 | 000 |

| PE2 | | | | | |
|-----|-----|-----|-----|-----|-----|
| 255 | 255 | 255 | 255 | 255 | 255 |
| 000 | 000 | 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 | 000 | 000 |

| PE3 | | | | | |
|-----|-----|-----|-----|-----|-----|
| 255 | 255 | 255 | 255 | 255 | 255 |
| 000 | 000 | 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 | 000 | 000 |

Figure 3.4.2d. Workspace arrays (6-by-6) after edge fixup operations.

| PE0 | | | |
|-----|-----|-----|-----|
| 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 |
| 170 | 170 | 170 | 170 |

| PE1 | | | |
|-----|-----|-----|-----|
| 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 |
| 170 | 170 | 170 | 170 |

| PE2 | | | |
|-----|-----|-----|-----|
| 085 | 085 | 085 | 085 |
| 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 |

| PE3 | | | |
|-----|-----|-----|-----|
| 085 | 085 | 085 | 085 |
| 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 |
| 000 | 000 | 000 | 000 |

Figure 3.4.2e. Smoothed subimage (4-by-4) replaces original subimage.

| PE0 | | | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 |
| 255 | 255 | 255 | 255 | 255 | 255 | 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 | 255 | 255 | 255 | 255 | 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | | | | | | | | 255 | 255 |
| 255 | 255 | 255 | | | | | | | | 255 | 255 |
| 255 | 255 | 255 | | | | | | | | 255 | 255 |
| 255 | 255 | 255 | | | | | | | | 255 | 255 |
| 255 | 255 | 255 | | | | | | | | 255 | 255 |
| 255 | 255 | 255 | | | | | | | | 255 | 255 |
| 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 |

Figure 3.4.2f. Workspace array (12-by-12) showing savings in number of pixels copied from original subimage.

final result is the "union" of the results written in Figure 3.4.2g and Figure 3.4.2h.

The smoothing algorithms appear in Appendix A3.1. The original algorithm is "smooth.orig.s" The improved algorithm is "smooth.impr.s" The reader should note the use of index registers in "stepping" through the image data arrays. The instruction sequence necessary to effect an interprocessor data transfer should also be noted.

For the unrealistically small image sizes that were used for the early simulations, the first smoothing algorithm presented appears to be more efficient than the second "improved" algorithm. This is true since the second algorithm has a more complex array indexing scheme but not an offsetting savings in the number of operations performed. An analysis of the "cross-over" point of these two algorithms will be presented later. An algorithm for which no original subimage copying is done is even more complex than the second smoothing algorithm presented, even though data copy operations are saved. This is because each border and corner pixel represents a special case to be handled by the algorithm, negating the time saved by not doing pixel copy operations.

Simulations of both algorithm versions were performed for a variety of image sizes ranging from 16-by-16 to 128-by-128 pixels. The complete image was superimposed onto an array of 4-by-4 (=16) PEs such that each PE processed a subimage of from 4-by-4 to 32-by-32 pixels.

Table 3.4.1 compares the simulation and timing results for the two smoothing algorithms. The run time has been normalized such that the original algorithm run time is equal to 1.00. These results indicate that the original algorithm is more efficient for small subimages (fewer than 12-by-12 pixels per PE) than the "improved" algorithm. The improved algorithm would be used for real-world-size problems.

The actual algorithm execution time can be calculated for a given algorithm/image size pair by multiplying the number of cycles by the cycle time. Assuming a standard 8MHz MC68000 processor, the internal cycle time is two clock cycle times, or 250ns. Thus, a 128-by-128 (=16K) pixel image can be smoothed by the 16-PE system in about 31ms. Note that this is algorithm execution time. The simulations do not include data load/unload time between primary and secondary memory (which will be highly implementation

| PE0 | | | |
|-----|-----|-----|-----|
| 255 | 255 | 255 | 255 |
| 255 | | | 255 |
| 255 | | | 255 |
| 170 | 170 | 170 | 170 |

| PE1 | | | |
|-----|-----|-----|-----|
| 255 | 255 | 255 | 255 |
| 255 | | | 255 |
| 255 | | | 255 |
| 170 | 170 | 170 | 170 |

| PE2 | | | |
|-----|-----|-----|-----|
| 085 | 085 | 085 | 085 |
| 000 | | | 000 |
| 000 | | | 000 |
| 000 | 000 | 000 | 000 |

| PE3 | | | |
|-----|-----|-----|-----|
| 085 | 085 | 085 | 085 |
| 000 | | | 000 |
| 000 | | | 000 |
| 000 | 000 | 000 | 000 |

Figure 3.4.2g. Smoothed results from operations on workspace arrays placed in result (4-by-4) array.

| PE0 | | | |
|-----|-----|-----|--|
| | | | |
| | 255 | 255 | |
| | 255 | 255 | |
| | | | |

| PE1 | | | |
|-----|-----|-----|--|
| | | | |
| | 255 | 255 | |
| | 255 | 255 | |
| | | | |

| PE2 | | | |
|-----|-----|-----|--|
| | | | |
| | 000 | 000 | |
| | 000 | 000 | |
| | | | |

| PE3 | | | |
|-----|-----|-----|--|
| | | | |
| | 000 | 000 | |
| | 000 | 000 | |
| | | | |

Figure 3.4.2h. Smoothed results from operations on original subimage arrays placed in result array (4-by-4).

Table 3.4.1. Comparison of smoothing algorithm simulation and timing characteristics. The "original" algorithm run time results are normalized to 1.00. The internal cycle time is 250ns. All of the simulations are performed with $N = 16$ PEs.

| TOTAL IMAGE SIZE (PIXELS) (PIXELS) | SUBIMAGE SIZE (PIXELS PER PE) | ORIGINAL ALGORITHM | | | | IMPROVED ALGORITHM | | |
|--|--|--------------------------|------------------|----------------------|--------------------------|--------------------|----------------------|--|
| | | INSTRUCTIONS EXECUTED | TIME (CYCLES) | TIME (NORMALIZED) | INSTRUCTIONS EXECUTED | TIME (CYCLES) | TIME (NORMALIZED) | |
| 16-by-16 | 4-by-4 | 729 | 400.2 | 1.00 | 796 | 4370 | 1.09 | |
| 32-by-32 | 8-by-8 | 2011 | 12588 | 1.00 | 2089 | 13079 | 1.04 | |
| 48-by-48 | 12-by-12 | 4161 | 26628 | 1.00 | 4060 | 26362 | 0.99 | |
| 64-by-64 | 16-by-16 | 6005 | 42198 | 1.00 | 5615 | 39875 | 0.94 | |
| 128-by-128 | 32-by-32 | 18443 | 146476 | 1.00 | 15493 | 123040 | 0.84 | |

dependent, e.g., see [SiS81b]). They also ignore operating system calls to effect network transfers: the network hardware is manipulated directly.

The 128-by-128 pixel simulation required about 16 minutes of VAX CPU time. This corresponds to an average execution rate of over 19 SIMD instruction per second of CPU time. Recall that the simulator executes a single PE instruction N (=16) times, once for each PE. Somewhat less than half of the CPU time may be saved if the "PE memory dump" following the simulation is inhibited. The writing of 128^2 numbers to disk files (for verification of the smoothed output) takes a considerable amount of time.

A "serialized" (single PE) algorithm was constructed from the original parallel algorithm to determine the speedup. The serial algorithm operates on the entire image (rather than a subimage) and thus does not need to perform masking or inter-PE transfer operations. A listing of the serialized algorithm appears in Appendix A3.1 as "smooth.ser.s." When the number of masking and transfer operations per pixel processed (parallel overhead) is high, the parallel algorithm will not be very efficient. If no overhead is involved, the parallel algorithm should execute 16 times faster on a 16-PE machine than on a 1-PE machine.

It was observed that the MC68000 divide instruction, which is executed once per pixel processed to scale the result, accounts for roughly 35% of the total run time. The divide instruction requires about 75 machine cycles as compared to a typical add instruction requiring about 4 cycles. If better run times were necessary, the algorithm could be restructured to smooth a window of eight nearest-neighbor pixels (as opposed to nine) and scale the data by shifting the result right by three bits. A typical 3-bit shift requires 7 cycles, or about 10% of the divide cycle time. However, a load and add cycle (about 7 cycles) is saved since only eight pixels are used in the window. Thus a 35% improvement can be gained by replacing the divide instruction.

The 75 cycles for a divide instruction is the maximum instruction time; the actual time required is data-dependent and is not determined by the simulator. If some PEs finish the instruction before others, they will be made to wait until all the PEs have finished. Recall that a PE instruction is dequeued from the FIFO buffer only when all PEs make the request for the next instruction. However, if all of the PEs finish the division before the 75 cycle

maximum, the hardware will be able to exploit this and release the next instruction to the PEs (since all PEs will be requesting the next instruction).

The simulation results presented may be extrapolated to determine timings and speedups for other machine and/or image sizes. The run time of an algorithm depends on the *relative* sizes of the machine and the image, or equivalently, the subimage size in pixels per PE. For the smoothing examples, a minimum machine size of 4 PEs is necessary and sufficient so that all relevant inter-PE transfer and masking instructions are included. For example, a 4-PE SIMD machine can smooth an 8-by-8 pixel image in the same amount of time as a 16-PE machine can smooth a 16-by-16 pixel image. In each case, a PE operates on a subimage of 16 pixels. Similarly, since 16 PEs can smooth a 128-by-128 (=16K) pixel image in 31ms, a 64-PE system of the same design and using the same algorithm could smooth a 256-by-256 (=64K) pixel image in 31ms. (For larger machines, the number of stages in the multistage cube network will increase; however, assuming that the propagation delay of the network is overlapped with PE operations, the impact of the added stages is negligible.) In general, increasing the number of PEs by a factor of four allows four times as many pixels to be processed in the same amount of time. However, this does *not* mean that processing four times as many pixels will take four times as long for a fixed machine size. In the latter case, the fixed and variable costs of performing the particular algorithm must be taken into account.

Table 3.4.2 shows the simulation results from the improved smoothing algorithm for $N = 16, 64$, and 256 and image sizes up to 64-by-64 pixels. As shown, speedup and efficiency increase as the subimage dimensions increase. Processor utilization is affected by two different mechanisms. For a fixed input image size, as N increases, the subimage size decreases. For smaller subimages, the number of pixels that are on subimage edges (and thus participate in inter-PE transfers) compared to the total number of pixels increases. This increases the proportion of time spent in overhead (transfers) and hence decreases speedup, efficiency, and utilization. The second mechanism is also influenced by N . As mentioned earlier, pixels on the edges of I are simply copied to O . This implies that PEs having image "edge" pixels in their subimages ("edge PEs") must perform these operations while "interior" PEs do not. Masking operations are used to enable the "edge" PEs and disable the

Table 3.4.2 Smoothing simulation SIMD speedup, efficiency, and utilization results.

| N | P = N * S | S = P / N | SPEEDUP | EFFICIENCY | UTILIZATION |
|-----|-----------|-----------|---------|------------|-------------|
| 16 | 64-by-64 | 16-by-16 | 14.32 | 0.90 | 0.86 |
| | 32-by-32 | 8-by-8 | 13.16 | 0.82 | 0.78 |
| | 16-by-16 | 4-by-4 | 9.52 | 0.60 | 0.57 |
| | 8-by-8 | 2-by-2 | 4.61 | 0.29 | 0.28 |
| | 4-by-4 | 1-by-1 | 1.85 | 0.12 | 0.11 |
| 64 | 64-by-64 | 8-by-8 | 13.16 | 0.82 | 0.73 |
| | 32-by-32 | 4-by-4 | 9.52 | 0.60 | 0.53 |
| | 16-by-16 | 2-by-2 | 4.61 | 0.29 | 0.26 |
| | 8-by-8 | 1-by-1 | 1.85 | 0.12 | 0.11 |
| 256 | 64-by-64 | 4-by-4 | 9.52 | 0.60 | 0.46 |
| | 32-by-32 | 2-by-2 | 4.61 | 0.29 | 0.22 |
| | 16-by-16 | 1-by-1 | 1.85 | 0.12 | 0.09 |

"interior" PEs for this copying operation. As N increases, the proportion of "edge" PEs to "total" PEs (N) decreases which reduces processor utilization.

Comparing the SIMD and MIMD implementations of the smoothing algorithm, the simulation indicates that the SIMD algorithm is only about 1.15 times faster than the MIMD algorithm. This is because there are relatively few control operations in the smoothing algorithm. Having a high ratio of MC to PE instructions in the SIMD algorithm means good speedups are attainable due to decoupling; the high ratio makes the equivalent MIMD algorithm much more costly to perform. Recall however that the MIMD time is underestimated; therefore, the difference between the MIMD and SIMD algorithm performance is more marked.

CHAPTER 5

HISTOGRAMMING ALGORITHM

Histogramming is commonly used for information extraction in image processing, e.g., it can be used to set thresholds dynamically for filtering or feature extraction. The SIMD histogramming algorithm given here is based on one described in [SiS81b]. The simulation results reported here were originally presented in [KuS84].

The algorithm accepts S pixels (picture elements) per PE, coded with G gray levels. The gray level of each pixel indicates how "dark" that pixel is where 0 means white and $G-1$ means black. Since the global histogramming algorithm ignores spatial information, the allocation of data to the N PEs is irrelevant except that each PE must contain the same number of pixels. The output of the algorithm is a count of the number of occurrences of each gray level or range of gray levels. The number of bins will be given by B . It is assumed that $B=2^b$, $G=2^g$, $N \geq B$, and $G \geq B$. Each PE will calculate a B -bin histogram based on the S pixels in its memory. Then these "local" histograms will be combined.

The local histograms are calculated by right-shifting each pixel (zero-fill) by $g-b$ bits to determine the appropriate bin to increment. For example, if $G=256$ and $B=4$, each pixel will be shifted right by 8 bits. Pixel values in the range 0-63 will be mapped to bin 0, 64-127 to bin 1, etc. The appropriate bin is incremented by one for each of the S pixels in each PE.

In the next b steps, each block of B PEs performs B simultaneous recursive doublings [Sto80] to compute the histogram for the portion of the image contained in the block (see Figure 3.5.1). In the first step, each block of PEs is divided in half such that the PEs with the lower addresses form one group, and the PEs with the higher addresses form another. Each group accumulates the

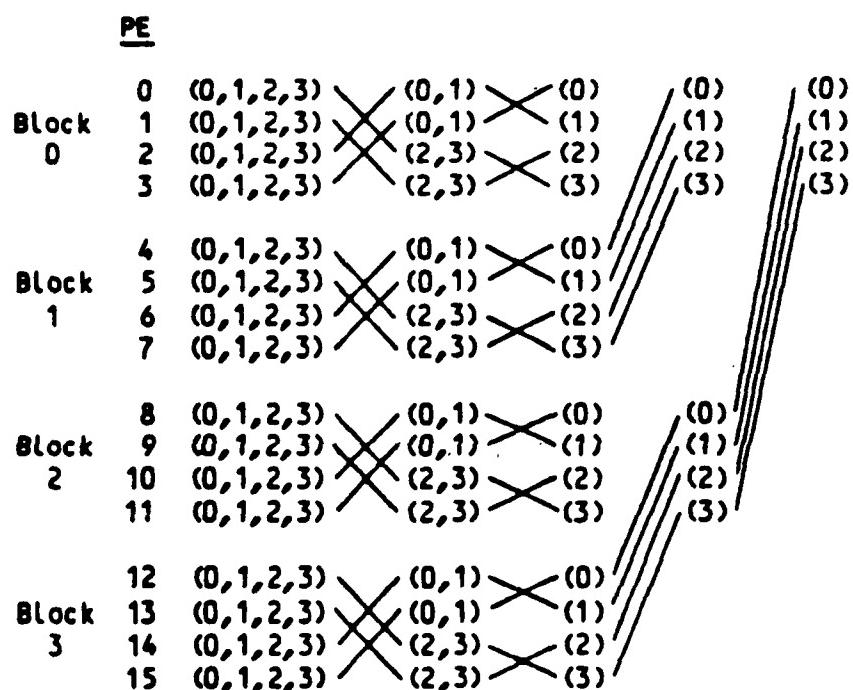


Figure 3.5.1. Histogram calculation for $N = 16$ PEs, $\text{nbins} = 4$ bins. (w, \dots, z) denotes that bins w through z of the partial histogram are in the PE.

sums for half of the bins, and sends the bins it is not accumulating to the other group. For example, in the first merging step, PE 2 sends its bin 0 value to PE 0 and PE 0 sends its bin 2 value to PE 2, simultaneously. PE 0 adds its kept bin 0 value to the bin 0 value received from PE 2, while PE 2 accumulates bin 2. Identical sequences of steps occur for corresponding pairs of PEs as shown in Figure 3.5.1. For each successive merging step, the groups of PEs are re-subdivided with the accumulated subtotals for each bin being combined into half as many PEs at each step. The subdividing process continues until there is one PE in each group and each PE has the total value for one bin from the portion of the image contained in the B PEs in its block. The next $n-b$ steps combine the results of these blocks to yield the histogram of the entire image distributed over B PEs. The sum for bin i will end up in PE i , for $0 \leq i < B$. This is done by performing $n-b$ steps of a recursive doubling algorithm to sum the partial histograms from the N/B blocks, shown by the last two steps of Figure 3.5.1. The recursive doubling steps are done for all B bins simultaneously. Each of the parallel inter-PE transfers required by this algorithm can be performed by a multistage cube network in one step [Sie85].

Simulations were performed for $N = 16, 64$, and 256 PEs. A 4-bin histogram of 8-bit pixels was performed on image sizes ranging from 2^4 to 2^{12} bytes. The results of the simulations and timings are given in Table 3.5.1. The histogramming algorithm code is given in Appendix A3.2 as "histo.s."

For a given image size, each time the number of PEs is increased by a factor of four, the subimage size per PE is reduced by a factor of four. However, the number of "overall machine cycles" (execution time in cycles) does not always decrease correspondingly. Using four times as many PEs speeds up the calculation of the local histograms by a factor of four (neglecting some additive constants) since all PEs are being fully utilized. When the local results are combined, each factor of four increase in the number of PEs results in two additional transfer/add steps. Together with the fixed MC overhead in initialization, these factors account for the less than optimum speedup when adding PEs. (An optimum speedup would be a factor of k when k times as many PEs are used.)

For some total image sizes, notably the 16-by-16 and 8-by-8 cases, the execution time of the algorithm actually *increases* when the subimage size per PE

Table 3.5.1 Histogramming simulation SIMD execution time results. The total image size varies from 4-by-4 to 64-by-64 and is distributed over $N = 16, 64$, and 256 PEs.

| N | $P = N \cdot S$ | $S = P / N$ | MACHINE CYCLES CU/PE/OVERALL | MAX QUEUE WORDS USED |
|-----|-----------------|-------------|---------------------------------|-------------------------|
| 16 | 64-by-64 | 16-by-16 | 4625/6900/7457 | 32 |
| 64 | 64-by-64 | 8-by-8 | 2369/2212/2897 | 32 |
| 256 | 64-by-64 | 4-by-4 | 1985/1124/1985 | 29 |
| 16 | 32-by-32 | 8-by-8 | 2129/2100/2657 | 32 |
| 64 | 32-by-32 | 4-by-4 | 1745/1012/1745 | 26 |
| 256 | 32-by-32 | 2-by-2 | 1829/824/1829 | 6 |
| 16 | 16-by-16 | 4-by-4 | 1505/900/1505 | 24 |
| 64 | 16-by-16 | 2-by-2 | 1589/712/1589 | 6 |
| 256 | 16-by-16 | 1-by-1 | 1790/749/1790 | 5 |
| 16 | 8-by-8 | 2-by-2 | 1349/800/1349 | 6 |
| 64 | 8-by-8 | 1-by-1 | 1550/637/1550 | 5 |
| 16 | 4-by-4 | 1-by-1 | 1310/525/1310 | 5 |

falls below 4-by-4. This is due to the relatively costly interconnection network set-ups and transfers required for the correspondingly small number of local results. Since the number of transfers is independent of subimage size, transfers will require higher percentages of execution time for small subimage sizes. Therefore, given the total image size, Table 3.5.1 suggests that for minimum execution time the number of PEs be chosen such that the subimage size per PE is about 4-by-4.

The algorithm behavior can be more easily understood by examining each of its three major steps individually. The first step is initialization and is done primarily by the MC. During this step, PEs are idled most of the time. The second step is the calculation of the local histograms. Both the MC and PE execute a number of instructions that is proportional to S, but the PE instructions required to process a pixel take longer than the loop counting and branching instructions executed by the MC. During this step, the PE instruction queue begins to fill up. Analysis of the algorithm has shown that for S greater than about 25 pixels, the queue fills to capacity. The third part of the algorithm involves the merging of the local histograms. The number of MC and PE instructions is a function of both B and N for this part, but now the MC instruction time required for loop counting and the construction of masks dominates the PE instruction time. Therefore, the PE instruction queue will be empty part of the time.

As the subimage size per PE decreases, the ratio of MC CPU to PE CPU machine cycles increases. Both the MC and PE execute fewer instructions in the local histogram calculation part of the algorithm, where the PE time dominates, but the same number of instructions in the initialization and merging steps. This explains the relative changes in the MC and PE execution times as seen in Table 3.5.1.

The "overall machine cycles" figure is always greater than or equal to the maximum of the MC and PE "machine cycles" figures. Regardless of subimage size, there are approximately 550 MC machine cycles that cannot be overlapped with PE instructions for this algorithm. These cycles represent initialization and some of the merging steps that the MC does when the PE instruction queue is empty. When the subimage size per PE is small, the "overall" figure is equal to the number of MC machine cycles; thus the MC was the

limiting factor in the algorithm. All of the PE operations were able to be completely overlapped with the MC instructions in spite of the MC cycles that could not be overlapped. As the subimage sizes became larger, the "overall" number became somewhat larger than either of the MC CPU or PE CPU figures. Here, the PE execution time is larger than the MC time. Again, PE operations were able to be completely overlapped with MC instructions through the use of the queue, but the overall time represents the PF time plus the 550 or so MC cycles that were not overlapped.

The "maximum queue size used" column of Table 3.5.1 indicates that the 32-word instruction queue assumed for these simulation runs was completely filled at some point for the larger subimages, but was never filled for the 4-by-4 and smaller subimages. The filling of the queue is desirable since it represents the greatest possibility for overlap of the MC and PE instructions. The 32-word length allows maximum decoupling for these simulation runs: the execution times in Table 3.5.1 did not decrease as the queue was lengthened. This conclusion is based on results from simulation runs for other queue length assumptions which are not presented here. All N-PE and 1-PE SIMD machine simulations given here have allowed for maximum decoupling.

A 1-PE SIMD algorithm was constructed from the original parallel algorithm to determine the speedup. The PE in this algorithm operates on the entire image rather than a subimage. Table 3.5.2 indicates the speedups for the range of machine and image sizes of Table 3.5.1. Here again, the subimage size per PE seems to have a large influence on the effectiveness of the parallel algorithm. The best speedup obtained was for the 16-by-16 subimage size (13.75 of an ideal 16.00), and better speedups would have been obtained for larger subimage sizes. While for the 64-by-64 subimage and 256-PE case the speedup is far from ideal, "real-world" applications may involve total image sizes as large as 5000-by-5000 and would provide much better speedups.

The speedup in Table 3.5.2 compares a 1-PE SIMD machine to an N-PE SIMD machine. Consider the histogram execution time on a single MC68000, which must perform both the control flow operations (previously done by the MC) as well as the arithmetic operations, and these operations cannot be overlapped. For comparison, the overall machine cycles required by an MC68000 to perform the 64-by-64 global histogram was 160,186 (calculated by simulation).

Table 3.5.2. Histogramming simulation SIMD speedup, efficiency, and utilization results.

| N | P = N * S | S = P / N | SPEEDUP | EFFICIENCY | UTILIZATION |
|-----|-----------|-----------|---------|------------|-------------|
| 16 | 64-by-64 | 16-by-16 | 13.75 | 0.85 | 0.61 |
| | 32-by-32 | 8-by-8 | 9.69 | 0.60 | 0.32 |
| | 16-by-16 | 4-by-4 | 4.35 | 0.27 | 0.11 |
| | 8-by-8 | 2-by-2 | 1.29 | 0.08 | 0.01 |
| | 4-by-4 | 1-by-1 | 0.45 | 0.03 | 0.00 |
| 64 | 64-by-64 | 8-by-8 | 35.41 | 0.55 | 0.19 |
| | 32-by-32 | 4-by-4 | 14.77 | 0.23 | 0.05 |
| | 16-by-16 | 2-by-2 | 4.13 | 0.06 | 0.01 |
| | 8-by-8 | 1-by-1 | 1.42 | 0.02 | 0.00 |
| 256 | 64-by-64 | 4-by-4 | 51.69 | 0.20 | 0.01 |
| | 32-by-32 | 2-by-2 | 14.10 | 0.05 | 0.00 |
| | 16 by-16 | 1-by-1 | 3.69 | 0.01 | 0.00 |

Dividing this figure by 7457 (the number of cycles required by a 16-PE SIMD machine, from Table 3.5.1), it is calculated that the 17-processor system configured as an SIMD machine achieves an absolute SIMD speedup of 21.48. The "ideal absolute SIMD speedup" is $2N$ or 32 in this case.

In MIMD mode, each PE would perform the same instructions, but there is no explicit synchronization among processors between instructions. Thus some 68000s may complete the local histogram calculation and their merging steps before others. Use of the interconnection network will be asynchronous; thus, some connections may be temporarily "blocked" due to contention at the network switches, which will delay the processing in some 68000s. Again using the total image size of 64-by-64, the processing time for a 16-processor MIMD system (where each processor is a single MC68000) was found to be 10511 cycles (by simulation). Dividing this by 7457, the overall machine cycles for the 16-PE (and 1-MC) SIMD case, it is determined that the SIMD algorithm performs the task at least 1.40 times faster than a comparable MIMD algorithm. This is a conservative estimate since the simulator underestimates the MIMD execution time.

The processor utilization for the SIMD histogramming algorithm is unity during the calculation of the local histograms. The first b merging steps involve all of the PEs, but due to the overhead from MC masking operations, the instruction queue is usually empty and the PEs idled. The utilization during these steps is about 0.04 for the 4-bin case. As the number of bins increases, the number that are transferred for each setting of the network increases, increasing efficiency. Utilization is also improved as the number of bins increases because the time involved in index calculations (for which only half of the PEs are enabled at a time) is fixed while the time involved in transfers (for which all PEs are enabled) is increasing.

Even if the overhead from masking and data transfers was ignored for the first b merging steps, the speedup and efficiency would still be non-ideal. This is due to the *redundancy* of the additions being performed. There are redundant computations if the N -PE algorithm requires more than N times the number of arithmetic computations as the 1-PE algorithm [SiS82b]. For the serial histogramming algorithm, P addition operations are required to update the bin counts. For the parallel algorithm, P/N addition operations are done

during the local histograming phase. Additional operations are required during the merging phases. Therefore, the parallel algorithm has more than N times the number of addition operations as does the serial algorithm.

During the final $n-b$ merging steps, half as many PEs participate at each step and there is considerable masking and transfer overhead. Hence the utilization is considerably less than in the previous steps. Increasing the number of bins increases efficiency and utilization since it reduces the number of merging steps that do not involve all N PEs. In fact, when $B = N$, utilization is maximized since all N PEs are always enabled.

CHAPTER 6

CONTOUR EXTRACTION

3.6.1 Introduction

Many individual image and speech processing algorithms and their formulations for parallel processing environments have been studied. However, rarely is a complete scenario considered as a whole. Consider the situation where the results of one algorithm are used as input to another. In the parallel environment, this may strongly influence how each algorithm is structured. For example, results calculated in one PE might need to be communicated to another PE for use in a later algorithm.

Contour extraction is a key tool for use in applications ranging from computer assisted cartography to industrial inspection. Two algorithms from a contour extraction task will be used as an application example for demonstrating the architectural features which must be provided by PASM in order to have an appropriate execution environment. It will be shown how computational attributes of a parallel implementation of this example SIMD/MIMD scenario influence the hardware design choices, including those features that would be desirable in a custom-designed VLSI chip set.

The two algorithms to be considered are *edge-guided thresholding (EGT)* and *contour tracing*. The EGT algorithm, discussed in Section 3.6.2, is used to determine the optimal threshold for quantizing the image [MiR81]. The contour tracing algorithm, considered in Section 3.6.3, uses the set of optimal thresholds to segment the image and trace the contours. These two parallel algorithms are based on those developed in [TuA83] and are summarized here since their processing demands are quite different from each other. As will be seen, the EGT algorithm is best suited for SIMD mode while MIMD mode will be used for the contour tracing algorithm. Also, the EGT algorithm will have

inter-PE communication needs that are different from the communication needs of the contour tracing algorithm. Other aspects will be discussed in Section 3.6.4. For this task scenario, the ability to partition PASM is not used; i.e., all N PEs are employed.

3.6.2 Edge-Guided Thresholding

Consider an M -by- M pixel *input image* to be processed by the two algorithms. The value of each pixel is assumed to be an eight-bit unsigned integer representing one of 256 possible gray levels. Using the PASM model, assume that the PEs are logically configured as a \sqrt{N} -by- \sqrt{N} grid, on which the M -by- M image is superimposed, i.e., each processor has an M/\sqrt{N} -by- M/\sqrt{N} *subimage*. For $M = 4096$, $N = 1024$, each PE stores a 128-by-128 subimage. Each input image pixel is uniquely addressed by its i - x - y coordinates, where x and y are the x - y coordinates of the pixel in the subimage contained in PE i .

The EGT algorithm consists of three major steps. First, the Sobel edge operator [DuH73] is used to generate an *edge image* in which gray levels indicate the magnitude of the gradient. A figure of merit which indicates how well a given thresholded gray level image matches edges in the edge image is then computed for every possible threshold. Finally, the maximum value of the figure of merit function is chosen to determine the threshold level. This is done for each subimage independently; thus, the threshold levels may differ from one subimage to the next. The complete EGT algorithm is most easily formulated as the "parallel algol" SIMD procedure given in Appendix A3.3 as "egt.pa." An alternative formulation is the Parallel-C "egt.c" version also in Appendix A3.3. Let the subimage SI be M/\sqrt{N} -by- M/\sqrt{N} and $SI(i, x, y)$ be a subimage pixel, where $0 \leq x, y < M/\sqrt{N}$, $0 \leq i < N$. The algorithm is performed for all of the subimages (all i) simultaneously.

Referring to the algorithm, the Sobel operators, s_x and s_y , represent weighted pixel value differences in the x and y directions, respectively. The value g at a given pixel represents the gradient at that pixel and these values form the edge image. The presence of an edge is indicated by high edge image pixel values. Next, the local maximum and minimum pixel values over a 3-by-3 window are determined for each gray level image pixel. Note that the

same image pixels necessary for the calculation of the gradient can be re-used for the determination of the local maximum and minimum. The center pixel of the 3-by-3 window is an *edge point* if the threshold is greater than or equal to the local minimum and less than the local maximum. Running sums of the edge image pixels (gradient values) corresponding to edge points at each threshold ('sumedge') and a count of the number of edge pixels for each threshold ('nedge') are updated in the innermost 'for' loop. In general, each PE performs this 'for' statement using a different 'localmin' and 'localmax' and thus performs the statements in the loop (updates the sums) different numbers of times. This implies that each PE has the capability of maintaining its own loop index values. PEs are disabled when they finish their looping since PEs must remain synchronized in SIMD mode. The total time to perform the innermost 'for' loop is the maximum time taken by any PE.

The mean for each threshold ('sumedge' / 'nedge') is known as the figure of merit ('merit') and is calculated in the final 'for' statement using the accumulated sums. High figure of merit values indicate better matches between threshold-generated boundaries and the edges detected by the Sobel operator. The gray level associated with the maximum value of the figure of merit function is chosen for image segmentation.

The EGT algorithm is particularly well suited for SIMD parallelism because all pixels are processed similarly. This aids the PE-to-PE communication necessary when a PE must process pixels not in its subimage (i.e., in a neighbor PE). All PEs will simultaneously request the same pixel relative to their subimages. For example, when processing begins (with the upper left corner subimage pixel) all PEs will request (from the PE to their upper left) the pixel immediately above and to the left of their upper left corner pixel (if this pixel is within the complete image). This transfer of data from upper left neighbors can occur for all PEs simultaneously. In the case of this algorithm, transmission delays incurred due to PE-to-PE data transfers can be overlapped with data processing to reduce total execution time. A total of $4*(M/\sqrt{N} + 1)$ parallel transfers are needed for a M-by-M pixel image. The candidate interconnection networks for PASM can support these parallel transfers from any neighboring PE.

Since PE-to-PE communications in MIMD mode require explicit synchronization between the two processors for each data transfer, SIMD mode transfers should be used to more efficiently provide each PE with the one-pixel-deep border points of its subimage (from its neighbors). However, once each PE has all of the data it needs to perform the EGT algorithm, the calculations could proceed in MIMD mode. While MIMD mode would make the execution of the innermost 'for' loop more efficient (since no PEs would be disabled), this advantage must be weighed against the extra time involved in switching from SIMD to MIMD mode and requiring that each PE perform its own control flow operations for the outer two 'for' loops. Control flow operations include initialization and incrementing of loop counters, evaluation of conditional expressions, and branching. These operations are performed by the MC in SIMD mode for the outer two loops and can be overlapped with the PE operations. The next step of the scenario is contour tracing.

3.6.3 Contour Tracing

A contour tracing algorithm using MIMD parallelism and based on the one given in [TuA83] is summarized in this section. Initially, each PE contains a threshold value, T , for its subimage which was calculated using the EGT algorithm of the previous section. The contour tracing algorithm has two phases. In Phase I, the PEs segment their subimages based on the threshold and all local contours (both closed and partial) are traced and recorded. In Phase II, the partial contours traced during Phase I are connected.

A *contour table* is constructed in each PE containing an entry for every partial or complete contour in its subimage. Each contour table entry contains bookkeeping information such as the threshold value which generated the contour and a pointer to the i-x-y sequence of the contour. Each PE also contains a *partial contour list*. This list has an entry for each partial contour containing the i-x-y coordinates of its two end points and a pointer to its contour table entry.

In Phase I there is no PE-to-PE communication. PEs each use their threshold level to segment their subimage. To create the segmented image for threshold T , subimage pixels which have a value greater than or equal to T are

assigned a value of one; otherwise, the pixels are assigned a value of zero.

Contour tracing begins by scanning rows of the segmented image beginning with the top row. Scanning stops when a pixel with value one is found which has a zero-valued neighbor to either side (or both sides). This pixel is marked as the *start point* of a new contour, and its i-x-y coordinates are stored. For *edge PEs*, i.e., those on the edge of the \sqrt{N} -by- \sqrt{N} "grid" of PEs, no image points lie beyond the edge; thus all points in the leftmost (rightmost) column of the subimage of the PEs in the leftmost (rightmost) column of the grid of PEs are potential start points. For all other left and right subimage edges, it is assumed that the pixel in the neighboring PE is one-valued so that spurious start points are not chosen. Bypassing a potential start point (e.g., a left subimage edge with a zero-valued neighbor in the PE to its left) is not a problem since (1) contours have multiple potential start points within the subimage and (2) the partial contours will be connected in Phase II regardless of the start point chosen.

The contour is traced in a *counterclockwise direction (CCW)* first if the start point has a one-valued point to its right and is traced in a *clockwise direction (CW)* first if the start point has a one-valued point to its left. If there are zeroes to both sides, the initial direction chosen does not matter. Consider the start point pixel as the center pixel of the 3-by-3 window in which direction '0' is east, '1' is northeast, and so on [Fre61]. The CCW algorithm is stated as follows. Beginning with the neighboring pixel in direction five and incrementing by 1 modulo 8 to determine the next pixel, look for a pixel which has a value of one. When one is found, store the direction, p, of this new pixel and append its i-x-y coordinate to the contour sequence. Treat this pixel as a new center point of the 3-by-3 window. Then continue by looking for the next pixel in the contour beginning with the pixel in position $(p + 5)$ modulo 8. Tracing continues until the start point or a subimage boundary (point of indecision) is reached. The CW algorithm is similar but scanning begins with the pixel in position zero and decrements by 1 modulo 8 to determine the next pixel. After a point is found, the pixel in position $(p + 3)$ modulo 8 is scanned. Horizontal edges which span a subimage are also recognized; however, they are treated as a special case since no start point would have been identified. An implicit assumption is that all contours to be traced define regions that have area.

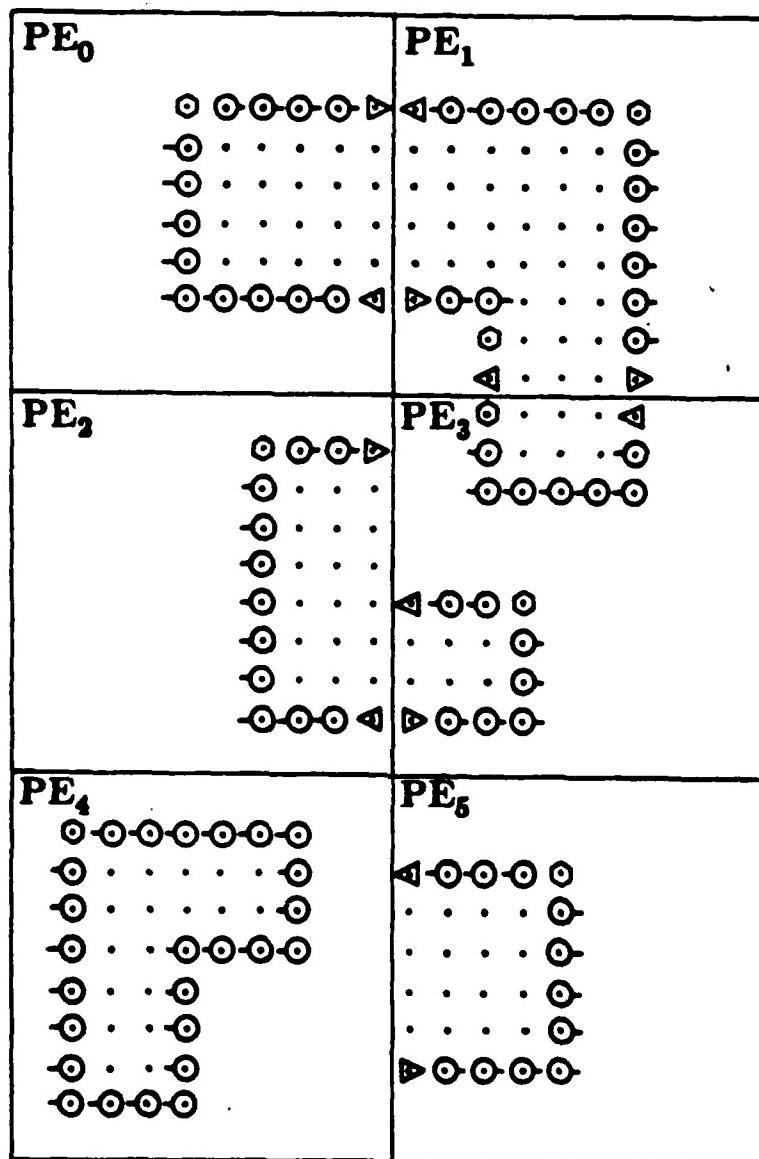
Examples of illegal contours that would not be traced are one-pixel-wide 'lines' or isolated points.

A *point of indecision* is reached when a pixel from an adjacent subimage would be required to determine the next direction of the contour [TuA83]. When a point of indecision is reached, it is recorded as an *end point*, and the algorithm returns to the start point to trace the contour in the opposite direction until another point of indecision is reached. When tracing in the CW (CCW) direction, the new contour pixels are inserted onto the front (back) of the i-x-y sequence. Pixels in the thresholded image are marked so that the contour will not be retraced.

As an example, a 30-by-20 image is divided into six 10-by-10 subimages; each subimage is loaded into one of six PEs. The result of Phase I processing is shown in Figure 3.6.1 where a dot indicates a one-valued pixel. Even though the entire object in PE 5 was located within the subimage, the left edge of the object was not traced in Phase I since PE 5 could not determine whether the object continued into the next subimage. On the other hand, a closed contour was found in Phase I for the object in PE 4 since the object did not include any border pixels of the subimage.

For the example of Figure 3.6.1, the local threshold value T is applied to the subimage in each PE. Each PE i begins scanning its respective subimage at pixel $(i, 0, 0)$ for a one (indicated by a dot) with a zero on either side. Depending on the start point found, tracing will proceed in either the CW or CCW direction first. For example, contours A, C, E, D, and G are traced in the CCW direction first while contours B, F, and H are traced in the CW direction first. In the example, PEs 1 and 3 have found two start points and have produced two traces. Once a PE has scanned the segmented image generated by its threshold, Phase I is complete.

In Phase II, each PE attempts to connect its partial contours to those located in neighboring PEs. In order for a PE to extend a contour, it must be able to access and modify contour tables which are located in other PEs. As a result, a mechanism to allow access to a contour table entry by only one PE at a time must be provided by the system and used by the contour tracing algorithm. A *semaphore* [Dij68] associated with each contour table entry is used to indicate whether or not that entry is locked, so no other processor can access it.



- Start point
- Counterclockwise trace mark
- Clockwise trace mark
- ◀ End point (counterclockwise)
- ▶ End point (clockwise)

Figure 3.6.1. Results of Phase I of contour tracing for a 30-by-20 subimage.
(Based on [TuA83]).

Semaphores are used to prevent variable access and updating problems due to interrupts. Details of these problems are beyond the scope of this discussion.

For the example of Figure 3.6.1, PE 0 might try to extend the CW end point of partial contour A by considering the possible extending pixels in PE 1 one at a time using the CW algorithm. To do this, PE 0 first locks the contour table entry for A. Then PE 0 requests that PE 1 check its partial contour lists to determine if any partial contour has the possible extending point as an end point. If such a partial contour exists, PE 1 locks the contour table entry pointed to by the partial contour list signifying that this entry is to be linked. In this case, PE 0 determines that A can be linked to B; thus, PE 1 locks B's contour table entry so that only PE 0 will be allowed to connect the partial contour. The i-x-y sequence for contour B is transferred to PE 0 and concatenated to the i-x-y sequence of partial contour A, forming a new, extended partial contour AB. If PE 0 found the contour table of partial contour B to be already locked, it will not be allowed to connect the contour. The extension of corner points is handled similarly, but involves communication with more than one PE. Note that the use of semaphores prevents another PE, i.e., PE 3, from using PE 1 to access B's contour table entry which PE 1 is in the process of modifying for PE 0.

Once PE i locates a partial contour in an adjacent subimage which continues the given contour and has stored the concatenated contour in its contour table, it repeats the process, if necessary, by following the contour to the next PE until the contour is closed or cannot be extended.

Independently of the actions of PE 0, PE 3 might attempt to extend contour D CCW to form the partial contour DC. If PE 3 attempted to extend the result, DC, when PE 0 is in the process of extending A into PE 1, it will find A locked. PE 3 then abandons its attempt to close the contour since PE 0 is also attempting to do it by unlocking partial contour DC. This allows PE 0 to access DC after it has appended B to A. Therefore, the closed contour ABDC is ultimately traced by PE 0. Alternatively, if PE 0 had completed linking B to A before PE 3 completed linking C to D, and PE 0 finds D locked, it would unlock AB. Thus, the closed contour would be traced by PE 3. Not allowing a PE to wait for another PEs locked contour table entry and requiring the blocked PE to unlock its affected partial contour prevents deadlock.

Occasionally, some contour tracing operations must be performed in Phase II before certain contours can be linked. Figure 3.6.2 shows a situation in which PE 2 traces contour E along the subimage boundary in Phase II before linking it to contour F. The subimage boundary pixels of contour H are also traced in Phase II.

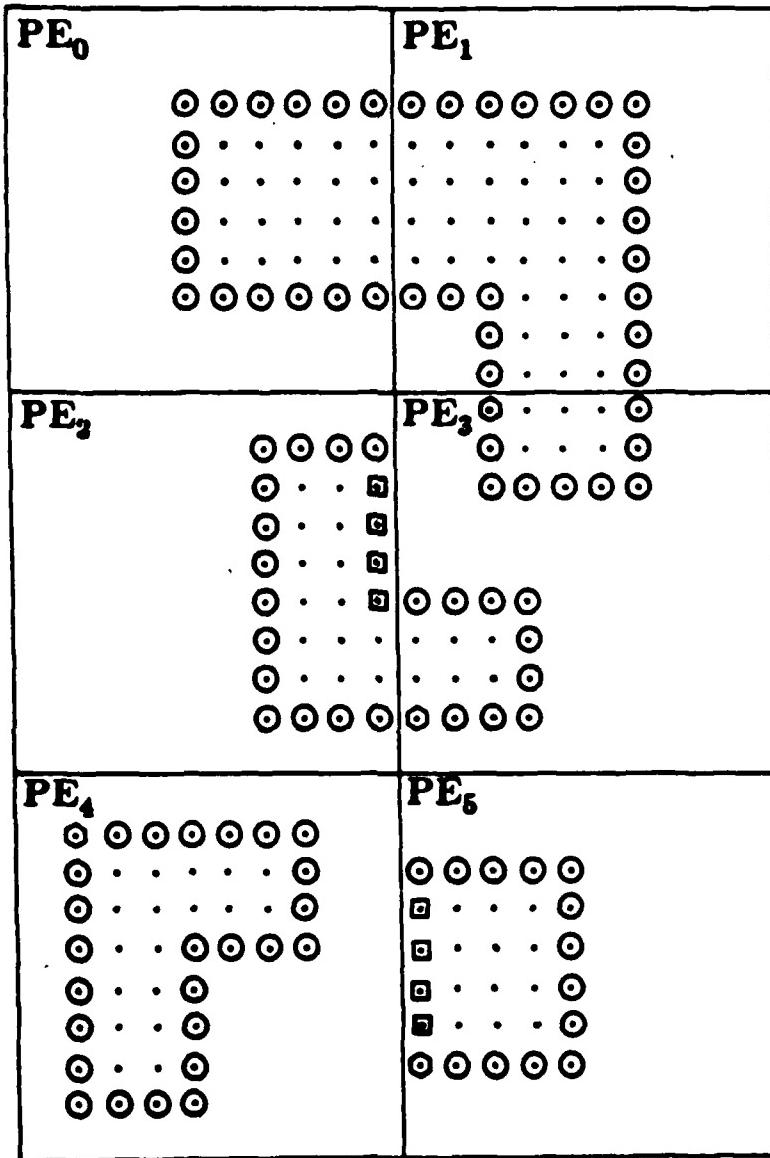
These examples demonstrate the basic ideas underlying the algorithms. The actual parallel algorithm details which ensure proper interaction of the PEs are complex and are not examined here.

When Phase II of the algorithm is complete, the i-x-y sequence for each contour in the image will be contained in exactly one of the PEs which contained part of the contour originally. The result of Phase II processing for the example of Figure 3.6.1 is shown in Figure 3.6.2. Since each PE tries to connect its contours independently, the number of the PE that finally closes a given contour is non-deterministic. While this may not be desirable in a few cases, in general the lack of a specific protocol determining which PEs can close contours equalizes both the processing load of each PE and the number of closed contours which eventually reside in each PE.

3.6.4 Architectural Implications

The study of a parallel image processing task leads to an understanding of necessary and useful hardware features for a system such as PASM. For the example algorithms, aspects of each which have an architectural impact will be listed. Processor-specific considerations (e.g., instruction set) are also treated because they can have a profound effect on the performance of the algorithms.

Although only two closely-related algorithms were presented in the previous sections, the two could hardly have been more different in their processing demands. As discussed in Section 3.6.2, the EGT algorithm is best suited for SIMD mode. This is because the algorithm requires data that is, by vast majority, local to each PE. Also, there are approximately (or exactly) the same number of pixels to be processed in each PE and all pixels are processed similarly.



- Pixels traced in Phase I
- Pixels traced in Phase II
- First pixel in the x-y sequence of the contour

Figure 3.6.2. Results of Phase II of contour tracing for a 30-by-20 subimage. (Based on [TuA83]).

When non-local data is needed in the EGT algorithm, the eight nearest neighbor PEs comprise the set of data sources. The PE-to-PE transfer of information must be efficient, or the parallel algorithms will be slowed. In its simplest form, this communication would be handled entirely by the PEs; each PE would control the network settings (through the use of routing tags [Sie85]) and perform all of the network protocol support (e.g., buffering, error detection). Each word transferred and each new network setting would require processor instructions. A more efficient method of PE-to-PE communication is by *direct memory access (DMA)*. DMA is a means by which data can be retrieved from one memory location and stored in another without processor intervention. The DMA hardware usually operates on a cycle-stealing basis so that a PE's access to its memory is not severely affected. In its basic form, PEs in SIMD mode would enter a DMA handling routine. This routine computes the local memory address range of the points to be transferred and sends this information to the special DMA hardware. The PEs then compute the destination address of the PE that is to receive the data and set the network accordingly. The DMA hardware then autonomously retrieves the information from local memory and performs the necessary network interfacing to send the data to the requesting PE. However, the PE would still be responsible for checking the incoming data (after the transfer is complete) for transmission errors, etc. A more advanced implementation of DMA capability is the use of an intelligent *Network Interface Unit (NIU)*. Requests for data from remote PEs would be made to the local NIU, which would interpret and satisfy the request by coordinating with a remote NIU. The NIU would combine DMA capability with network protocol support. VLSI technology may allow ready fabrication of sophisticated NIUs.

As discussed in Section 3.6.2, M/\sqrt{N} pixels come from each of four neighbors. For sake of example, let $M = 4096$, $N = 1024$, and $M/\sqrt{N} = 128$. Rather than involving the "source" and "destination" PEs in the individual transfers of these points, one of the DMA modes just described would be of great use. If pixels were stored in PEs by *row* (rows numbered 0-127) and the transfer from PE i to PE $i + \sqrt{N}$ was selected, the DMA hardware of PE i would be instructed to transfer 128 pixels starting at the address of "row 127" of the image. The DMA hardware associated with PE $i + \sqrt{N}$ would be set to

read 128 pixels from the network and store them beginning at an address representing "row -1." When data is transferred from a PE i to PE $i+1$, the situation is more complicated in that image data to be transferred is not contiguous. Conventional DMA hardware only supports physical block transfers of data. Here, a strong case for an intelligent NIU is made: the NIU could accept more complicated instructions such as "transfer 128 pixels starting at address X, taking every 128th pixel."

Floating point and special arithmetic function (e.g., square root, trigonometric) capability abounds in the form of "co-processor" chips. While the EGT algorithm involved only one "special function" (square root) in the calculation of the gradient, other algorithms such as image rotation, parallel root finding, and FFTs for speech processing make heavy use of these special functions. Since many of the special arithmetic functions are calculated using iterative procedures, a strong case is made for including hardware to perform these operations rather than to perform them in software. Software procedures where the number of iterations required is data-dependent are especially troublesome in SIMD mode since processors must be disabled as they complete the desired number of iterations. Also, the total time to perform an operation is the maximum time required by any processor (since the PEs are synchronized). There is a slight advantage to having special-purpose arithmetic functions on the same standard CPU chip in that data to be processed need not be moved between the two devices. VLSI technology should make such combined CPU-specialized arithmetic processor chips a reality.

The EGT algorithm has been simulated for N ranging from 16 to 256 and a total image size of 64-by-64 pixels. Although the details of the simulation results are not presented here, the general trends of the results will be described.

As the number of PEs (N) decreases, the subimage size increases since a fixed-size total image of 64-by-64 pixels was used. For large subimages, the ratio of subimage edge pixels to total subimage pixels is low, making processing very efficient. This is because inter-PE transfers make up only a very small fraction of the total processing time. A speedup approaching N was obtained for arithmetic operations for this case. As N was increased to 256 PEs, the subimage size decreases to 4-by-4. Here, the ratio of subimage edge pixels to

total subimage pixels is very high and inter-PE transfers make up a large percentage of the total processing time. While the total processing time is minimized as N increases, the speedup factor decreases. The simulations imply that N should be as large as possible for the EGT algorithm to minimize the processing time. However, this will make contour tracing (the next algorithm of the scenario) very inefficient since few contours will be traced in Phase I and heavy use of inter-PE communication will be needed to close the contours in Phase II. Thus the scenario must be considered as a whole, rather than as a sequence of individual algorithms.

Turning now to the contour tracing algorithm of Section 3.6.3, it is noted that both phases of the algorithm are suited to MIMD mode since they involve data-dependent execution times. Phase I of contour tracing requires only local data, while Phase II makes heavy use of non-local data. Phase I imposes no extraordinary requirements on the system since there are no special arithmetic operations and no network transfers to be done. Phase II however, with its somewhat arbitrary one-to-one connections (when transferring partial contour information between non-adjacent PEs), use of semaphores, and special signaling protocols imposes many new architectural requirements.

The interconnection network and any DMA or NIU hardware will be heavily used in Phase II processing when PEs extending partial contours probe remote PE memories that may contain the extensions of the partial contours. As in the EGT algorithm, NIU hardware would be of great use since it could process queries about possible extensions to partial contours without interrupting the remote PE. There will be a combination of short and long messages between PEs during this phase. A short message will occur when a PE extending a partial contour requests information about possible extending pixels from a remote PE. If a connecting partial contour is found, a long message consisting of the i-x-y sequence of the partial contour will be sent. Thus the interconnection network should support a variety of message sizes so that the efficiency of sending either type of message is high.

Since semaphores play a large part in ensuring correct linking of partial contours in Phase II, processors must be equipped with "test-and-set" or similar operations to facilitate a correct semaphore implementation. Most modern microprocessors already have some semaphore capabilities.

If the system is to support the execution of the two example algorithms well, it must be capable of dynamically switching between SIMD and MIMD operation, as can PASM. With only SIMD capability, the contour tracing algorithm would be executed with huge inefficiencies since there will be varying numbers and lengths of contours and arbitrary one-to-one communication patterns. A machine having only MIMD mode would be less seriously affected, but will lengthen execution time for the EGT algorithm due to the need for explicit synchronization for each data transfer step and the overhead of loop counter processing which is done concurrently by the MCs in SIMD mode. Thus, the capability to dynamically switch between SIMD and MIMD modes is important so that each algorithm can be executed in the most appropriate mode of parallelism.

Since PASM is an SIMD/MIMD system, the interconnection networks proposed for PASM will be capable of operating both synchronously and asynchronously. The proposed networks are of the multistage type and can perform both the nearest-neighbor and arbitrary one-to-one connections.

CHAPTER 7

RASTER-TO-VECTOR CONVERSION

3.7.1 Introduction

Two key steps in many pattern recognition tasks are image thinning and vectorization. Parallel forms of image thinning and vectorization algorithms are developed and analyzed in the context of a complete image raster to vector conversion process. Two image thinning algorithms are presented: a "peeling" algorithm using Arcelli's masks and the other a hybrid scheme that combines a "distance-measurement" and "peeling" algorithm to achieve a better average performance. The vectorization algorithm scans a thinned binary image, identifies line junctions and endpoints using template matching, and performs vectorization of curves using Ramer's polygonal approximation algorithm. These two algorithms are good candidates for parallelism because the image pixel data can be easily distributed among processors and because each pixel in the image must be repeatedly examined. While the parallel algorithms are more complex than their serial counterparts, they are shown to offer the possibility of improved execution time. Furthermore, interprocessor communication needed for these algorithms is limited to nearest-neighbor communication which allows a variety of parallel machines to use them.

Here, an *image raster* refers to a rectangular tessellation in which pixel positions are identified by a (row, column) coordinate. Each pixel has an intensity value (grey level). The first step in a raster to vector conversion task is *segmenting* image pixels into two classes: "object" and "background." Histogramming is used to determine the distribution of grey levels in the image. Depending on the characteristics of the histogram, the image may be able to be segmented by simple thresholding, by an adaptive technique such as edge-guided thresholding [Mil79, SuR82], or by more complex classification methods

[SwT81]. Parallel algorithms for histograming [KuS84, SiS81b], edge-guided thresholding [TuA83], and other methods [SiS81b] have been studied. The result of such a segmentation is a *binary image raster* where the 1-valued points correspond to "object" and the 0-valued to "background."

The application dictates whether "outlines" of objects or "skeletons" of objects are to be found. Both transformations reduce the amount of data to be stored by characterizing the shape of the features in the image. Generally, this simplifies later procedures used for recognition and classification of features. A parallel contour tracing algorithm [TuA83] can be used to find outlines. Thinning algorithms [ArC75, Hil83] are used to reduce elongated objects to one-pixel thick figures. In either case, it is often desirable to reduce the resulting curved "lines" to a set of *vectors* that form a piecewise-linear approximation of the original curve. The vectors need not match the curve exactly; they may be an estimation of it assuming some tolerance, e.g., ± 1 pixel.

Two image thinning algorithms are presented; a "peeling" algorithm using Arcelli's masks [ArC75] and the other a hybrid scheme that combines Rosenfeld's "distance-measurement" [RoP66] and the peeling algorithm to achieve a better average performance. The vectorization algorithm scans a thinned binary image, identifies line junctions and endpoints using template matching, and performs vectorization of curves using Ramer's polygonal approximation algorithm.

3.7.2 Line Thinning

Depending on the scan resolution of a map, picture, or camera image, lines may be from one to many pixels "thick." Practical vectorization algorithms require one-pixel-wide lines as input, thus, line thinning is an important step in the conversion process.

Two basic types of algorithms are being employed to thin lines. *Distance measurement* is a thinning algorithm that determines each pixel's distance from the edge of the line to which it belongs. Pixels with locally maximum distances are retained in the thinned line; non-local maximum pixels are removed [RoP66]. On the first pass, the image is scanned row-wise from the upper left-hand corner to the lower right-hand corner. Each 1-valued pixel in the image I

is assigned a new value according to:

$$I(\text{row}, \text{column}) = \min(I(\text{row}-1, \text{column}), I(\text{row}, \text{column}-1)) + 1$$

which calculates each pixel's distance from the left or top side of a line. On the second pass, the image is scanned row-wise from the lower right-hand corner to the upper left-hand corner. Each nonzero-valued pixel is assigned a new value according to:

$$I(\text{row}, \text{column}) = \min(I(\text{row}+1, \text{column})+1, I(\text{row}, \text{column}+1)+1, I(\text{row}, \text{column}))$$

which calculates each pixel's distance from the right or bottom side of a line, or the previously-calculated distance from the left or top side if shorter. A third pass identifies and sets to zero non-local maxima points that do not cause the line to become disconnected. A point is not a local maxima if $I(\text{row}, \text{column})$ has a neighbor with value $I(\text{row}, \text{column})+1$. Templates (3-by-3 windows) are used to determine if the line would become disconnected if the center point were removed. When the resulting image is used in later processing steps, pixels that are non-zero are taken to be 1-valued.

Peeling is another thinning algorithm in which pixels are “peeled” (removed) from line edges until a one-pixel-wide line remains. Arcelli’s algorithm [ArC75] peels pixels by comparing each 1-valued pixel and its neighbors in the image with a set of templates. The templates used to thin 8-connected lines are shown in Figure 3.7.1 [Hil83]. In the templates, zeros must match 0-valued pixels, ones must match 1-valued pixels, and asterisks can match either 0- or 1-valued pixels in the image. The algorithm works on two images, the “current” image to which it compares templates and a “working” image of the same size which it updates when templates are matched. Initially, the current image and the working image are identical copies of the original input image. To begin, template A1 is compared with all 1-valued pixels and their neighbors in the current image. If a match is obtained, the corresponding central pixel of the working image is deleted (changed to a 0-valued pixel). After processing with template A1, the current image is discarded, the working image becomes the new current image, and a new working image is obtained by copying the new current image. The process is repeated with template B1, then with A2, B2, A3, B3, A4, and B4, in that order, forming a complete cycle. When no pixels are removed during the processing of a complete cycle, the procedure ends.

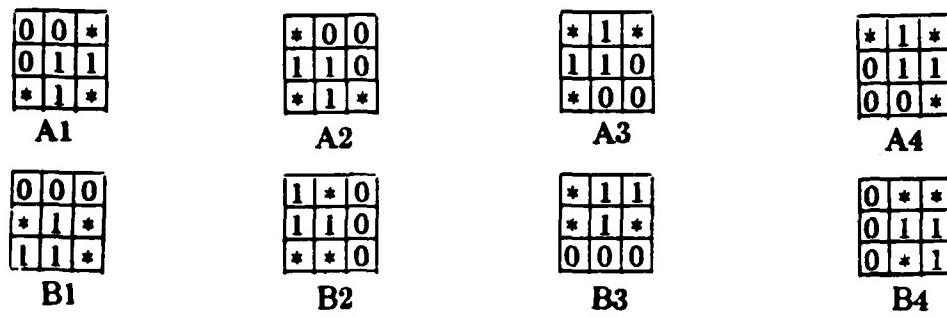


Figure 3.7.1. Arcelli's templates for thinning images (Based on [Hil83]).

For a fixed image size, the processing time for the peeling algorithm is line-width dependent because thick lines require more passes over the data while the distance algorithm requires a fixed amount of time. The distance algorithm is much faster than the peeling algorithm due to the smaller number of operations involved for each 1-valued pixel. Both thinning algorithms leave extraneous pixels, i.e., those that could be removed without altering connectivity or shortening lines. The peeling algorithm leaves very few extraneous pixels, usually at complicated line junctions. The distance algorithm often leaves lines that are two pixels in width, which is unacceptable without further processing. A more sophisticated peeling algorithm which employs additional templates [Hil83] to remove all extraneous pixels has been programmed for the CLIP-4 SIMD computer. Its execution time is slower than Arcelli's original procedure because many more templates must be applied.

Parallel Distance Algorithm

Consider the implementation of the distance algorithm on an N-PE machine logically arranged as an array of \sqrt{N} -by- \sqrt{N} PEs as shown in Figure 3.4.1. Each PE stores an M/\sqrt{N} -by- M/\sqrt{N} *subimage* block of the original M -by- M image I . Specifically, PE 0 stores the pixels in columns 0 to $(M/\sqrt{N})-1$ of rows 0 to $(M/\sqrt{N})-1$, PE 1 stores the pixels in columns M/\sqrt{N} to $2(M/\sqrt{N})-1$ of rows 0 to $(M/\sqrt{N})-1$, and so on.

During the first pass of the distance algorithm, the calculations for a 1-valued pixel at coordinate (row, column) depend on the values of the pixels above and to the left of it. Because each PE has only the data for its own subimage in its memory, results need to be transmitted as they are calculated to neighboring PEs below and to the right. Whenever a PE obtains a result for a pixel in the last column of a subimage, the result is passed to the PE on its right. By analogy, results obtained for pixels in the last row of a subimage are passed to the PE below. For the second (reverse) pass, results are passed to PEs above and to the left.

The straightforward implementation of the parallel distance algorithm is inefficient since at most \sqrt{N} PEs can be doing calculations at once for the forward or reverse distance-calculating passes. Each pass requires M^2 calculation

steps for a serial processor and

$$\frac{M^2}{\sqrt{N}} - \frac{M}{\sqrt{N}} + M$$

calculation steps for an N-PE machine. This can be derived by calculating the time (step number) at which each pixel is processed. If PE 0 begins processing the pixel in row 0, column 0 at time 0, it processes the pixel in row 0, column $(M/\sqrt{N})-1$ at time $(M/\sqrt{N})-1$ and the pixel in row 1, column 0 at time M/\sqrt{N} . Also at time M/\sqrt{N} , PE 1 may begin processing the pixel in row 0, column M/\sqrt{N} because the results from the PE on its left (PE 0) have been obtained. It can be seen that the pixel in row $M-1$, column 0 is processed at time $(M/\sqrt{N})(M-1)$ and the pixel in row $M-1$, column $M-1$ is begun $M-1$ time units after this (and completed after an additional time unit). This yields the result stated above.

In the best case with $N=M^2$ PEs (one pixel per PE), each pass would take $2M-1$ steps. The computational *speedup* (1-PE computation time / N-PE computation time) for this case would be

$$\frac{M^2}{2M-1} \approx \frac{M}{2} = \frac{\sqrt{N}}{2}$$

Because the ideal computational speedup is N , this indicates that the PEs are being used inefficiently. The speedup for practical numbers of PEs ($N \ll M^2$) is even worse.

Better speedups can be realized by allowing PEs to individually process pixels that do not depend on previously-calculated results from PEs above and to the left of them (or below and to the right). For example, note that 0-valued pixels remain 0-valued regardless of their neighbors and 1-valued pixels such as points A and B in Figure 3.7.2 remain 1-valued since one of the pixels above or to the left of them is known to be 0-valued.

An improved algorithm is stated as follows. Using the data allocation discussed earlier, each PE begins by passing its rightmost subimage column to the PE on its right and its bottommost subimage row to the PE below it. Transfers between different PEs occur simultaneously; e.g., when PE $J-1$ sends its upper right corner pixel to PE J , PE J sends its upper right corner pixel to PE $J+1$, PE $J+1$ sends its upper right corner pixel to PE $J+2$, etc. Cube-type networks can perform these transfers in a single step. After the $2(M/\sqrt{N})$

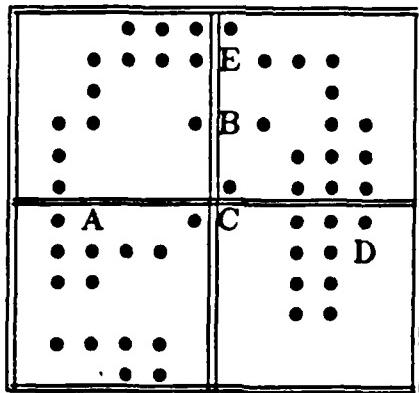


Figure 3.7.2. Enhanced distance algorithm; 1-valued pixels are denoted by dots or letters. Each of the four PEs has a subimage of six by six pixels. Pixels A and B can be processed immediately; C, D, and E require results calculated by other PEs.

transfers (the bottom right-hand corner is transferred twice), PEs independently calculate distances wherever they can do so. Points such as C, D, and E (Figure 3.7.2) require results from other PEs in order to be calculated; they are placed in a "do later" list in that PE. As described earlier, when results are obtained for rightmost columns or bottommost rows, these are passed to the PE to the right or below. Only nonzero-valued pixels need to be passed. As each updated result is received, items from the "do later" list are removed and calculated. An analogous procedure is performed in the reverse direction. For sparse images (relatively few 1-valued points and/or thin lines) the performance of this scheme approaches an ideal speedup.

A part-SIMD, part-MIMD mode algorithm is the best formulation of the improved algorithm: the initial passing of border points is done in SIMD mode while the calculations and passing of updated values are performed in MIMD mode. SIMD mode is preferred for the initial communication of border information because the interconnection network transfers are accomplished with explicitly programmed steps in the SIMD program which are executed synchronously. This is more efficient than asynchronous (MIMD) transfers because the destination PEs are not interrupted when incoming data arrives (eliminating the overhead due to interrupt-handling software) and there is no possibility of interconnection network conflicts. A conflict can occur when two or more data items wish to use the same internal network switching element or data link simultaneously. MIMD mode is preferred for the calculations and communication of updated values because it provides the flexibility to skip certain calculations and return to them later when the data becomes available. The ability of PASM to dynamically switch between the SIMD and MIMD modes of parallelism is used in this case to most efficiently perform each part of the algorithm.

The algorithm performance also depends on N , the number of PEs. For a fixed input image size, as N increases, the subimage size decreases. In general, the algorithm execution time will decrease as N increases because each PE processes fewer pixels. However, for smaller subimages, the ratio of pixels that are on subimage edges (and thus participate in inter-PE transfers) to the total number of pixels increases. This increases the proportion of time spent in overhead (transfers) and hence decreases the speedup. The optimal number of PEs is data dependent: larger N for sparse images; smaller N for less sparse images

to reduce the number of points placed on the "do later" lists. In the worst case, the amount of parallelism available to the straightforward and the improved versions of the distance algorithm is comparable. However, the actual worst-case performance of the improved algorithm may be poorer than that of the straightforward algorithm due to the overhead in handling the "do later" lists.

The final pass of the distance algorithm involves the determination and removal of non-local maxima. To begin this pass, PEs exchange subimage border information in SIMD mode as shown in Figure 3.4.1. Then in MIMD mode, each nonzero-valued pixel is examined. If it is not a local maxima and its removal would not cause disconnection of the line, it is removed from the image. The speedup here is data-dependent: it approaches N for those images that result in an equitable distribution of the workload among the PEs.

Parallel Peeling Algorithm

The peeling algorithm uses the same data allocation and border information from adjacent PEs as shown in Figure 3.4.1. After the exchange of border information, each PE thins the lines in its subimage independently. PEs begin at the upper-left-hand corner of their subimages and scan along the columns of row 0, then row 1, and so on. Each time a 1-valued pixel is encountered, a template is applied to determine whether the 1-valued pixel can be removed. The templates were shown in Figure 3.7.1. The first set of templates matches pixels that are on the "top" of lines; the second set matches pixels that are on the "right side" of lines; and so on. The templates are applied in series, causing wide lines to be stripped of one layer of pixels at a time. After the complete cycle of templates (one layer of pixels removed from all "sides" of lines), PEs again exchange border information and the complete process begins again at the upper-left hand corner of the image. The communication of border information is required after each cycle because a border pixel removed by a PE in one cycle will affect the results an adjacent PE calculates in the next cycle. The process ends when no more pixels can be removed.

The algorithm can be structured for all-SIMD, all-MIMD, or part-SIMD, part-MIMD mode processing. At first glance, it would seem that the all-SIMD

approach would be very inefficient since there are a differing number of 1-valued points in each PE, they occur at different locations within the subimage, and the templates need to be applied only at the 1-valued points. Because it is likely that *some* PE will have a 1-valued pixel at each local subimage coordinate and because PEs are completely synchronized in SIMD mode, PEs with 0-valued pixels will be idle while the PEs with 1-valued pixels apply the templates at each step. (In the MIMD algorithm, PEs would independently skip over 0-valued pixels.) However, in SIMD mode the PEs need not perform the control instructions needed to match the template, as would be required for the MIMD algorithm. (Control instructions include loop counting, branching, and local index calculations.) This is because the CU would perform these tasks, overlapping its operation with the PEs' computations. Furthermore, the exchange of border information is performed efficiently in SIMD mode using synchronized interconnection network transfers.

The protocol for communication and achieving synchronization between cycles for the all-MIMD case is as follows. As each PE completes the processing for a cycle, it transmits all of its border pixels to its neighbors. When a PE has received border pixels from all of its neighbors, it may begin processing the next cycle. When a PE completes a cycle that removed no pixels, it sends all of its neighbors a message indicating such and stops. Neighboring PEs will then know that the border points received from that PE are final results and will not change. Final results can be used over and over by the neighboring PEs that are still working. The algorithm is complete when all PEs stop.

In the part-SIMD part-MIMD scheme, the computation during each cycle is performed in MIMD mode and the communication is done in SIMD mode. PEs synchronize after each cycle and revert to SIMD mode to exchange border pixels.

The all-SIMD approach performs best for densely packed lines because a large fraction of the PEs are doing useful work at each step, the CU and PE operations may be overlapped, and the SIMD communication efficiency is better than that of MIMD. Sparse images in which the data are evenly distributed among the PEs are best processed by the part-SIMD part-MIMD algorithm. It combines the advantages of applying templates to only the small number of 1-valued points, high communication efficiency, and short waiting

times encountered by PEs waiting to synchronize between cycles. The all-MIMD approach works best for sparse subimages with disparate numbers of object pixels.

Parallel Hybrid Thinning Algorithm

Analysis of these parallel thinning algorithms prompted the development and testing of a "hybrid" algorithm consisting of both the distance and peeling approaches. In the hybrid algorithm, the "distance" algorithm would be used to locate the approximate "centerline" and remove the bulk of the unwanted pixels in a fixed number of passes. Then, the simple [ArC75] or extended [Hil83] peeling algorithm would be used to remove the remaining extraneous pixels. Because the peeling algorithm removes only one pixel layer at a time, it is inefficient in the early passes when lines are still thick. Therefore, the use of a more efficient algorithm for removing pixels in the early passes achieves a higher average performance. Unless *a priori* knowledge of the image characteristics dictate use of a certain thinning algorithm, the hybrid approach is preferred.

3.7.3 Parallel Vectorization

Many computer image processing algorithms require that objects in images be represented by sets of straight line segments (vectors). Vector format is conceptually more familiar, commonly used for display devices, and requires significantly less storage space.

The vectorization algorithm described here assumes a thinned, binary image is available. If the thinning was not ideal so that some thick junctions and spurious points remain, the algorithm will produce extra short lines. Gap removal, kink straightening, and other "beautifying" algorithms are not considered here. The data allocation among PEs is just as described for the thinning algorithm; therefore, each PE can process the same subimage for the complete raster to vector task.

The vectorization algorithm consists of three main elements: line end/junction identification and line following [Peu81], topological

reconstruction [Peu81], and iterative polygonal approximation [Ram72]. Line ends/junctions are identified by applying a template at each 1-valued point. A data structure which describes the graph model (vertices and edges) of the lines in the image is constructed. Finally, sets of vectors that approximate the lines in the image are obtained.

Line End/Junction Identification and Line Following

PEs begin with their thinned binary subimages and subimage border data of each of their neighbors. If the same PEs that performed the thinning are being used, each PE already has the necessary data. A PE's subimage with all of its neighboring borders is called an *augmented subimage*.

A set of templates has been constructed to allow the rapid identification of line ends and junctions. Each template is a 3-by-3 pixel window with a 1-valued center pixel and the other 8 pixels either 0- or 1-valued. Thus there are $2^8=256$ different templates, each representing one of the following cases: (1) line end, (2) line junction, (3) middle point, or (4) indeterminate point. Figure 3.7.3 shows examples of each type of template for the 8-connected case. Figure 3.7.3a is an example of a line end. The line junction (Figure 3.7.3b) is defined as the meeting point of three or more distinct lines. The template of Figure 3.7.3c matches a point on a line that is neither an end or a junction. Figure 3.7.3d is a point that could be either a junction or a middle point; however, a 3-by-3 template is insufficient to determine the type. At this stage of the algorithm, indeterminate points are treated as junctions. Later, indeterminate points that are adjacent to other indeterminate points and junctions are examined and re-classified as middle points or true line junctions.

Line end/junction identification and line following are done in a single pass using the following procedure. In MIMD mode, each PE scans its thinned binary subimage by row starting from the upper left-hand corner. When a 1-valued pixel is encountered in the subimage, scanning stops and the pixel is treated as the center point of a 3-by-3 window. This pixel is called the *start point*. Data from the augmented subimage are needed to apply the template if the start point is on the edge of the subimage. The template is used to classify the start point as an end, junction (or indeterminate), or middle.

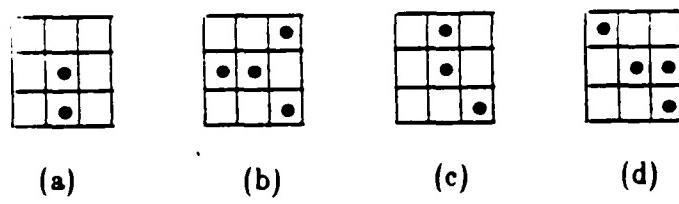


Figure 3.7.3. Templates for identification of line ends and junctions for 8-connected imagery: (a) line end; (b) line junction; (c) middle point; (d) indeterminate point.

If the start point is classified as an end, the line is followed until another end or junction is encountered. For "middle" points, the line is followed in each direction until ends or junctions are encountered. Junction (or indeterminate) start points have several lines emanating from them. Each line is followed until ends or junctions are encountered. When a line is followed, the coordinates of all 1-valued pixels in the line are recorded. Also, each point encountered is marked so it will not be retraced. This is especially important when tracing closed figures; if the start point is re-encountered, its classification is changed from "middle" to "closed figure entry point."

Due to the data allocation among processors, PEs will encounter lines that continue into an adjacent PE. With the augmented subimage data, PEs can determine whether a line stops on the edge of their subimage (a 0-valued pixel in the augmented subimage) or whether it continues (a 1-valued pixel in the augmented subimage). When a line continues into another PE, the pixel on the subimage edge is known as a "border connection point."

The desired output of the vectorization algorithm is a two-level structure. At the *graph level*, an edge and vertex representation of the image is formed. The graph's edges are the lines that emanate from vertices (end points, junctions, and closed figure entry points). This representation describes which vertices are adjacent (connected to each other by a single edge) and is useful for pattern matching or analysis of the connectivity of lines. At the more detailed *vector level*, each graph edge is modeled by vectors which give a piecewise-linear representation of the image line. This level is required for cartographic analysis and visual interpretation.

The graph level of the data structure is built as follows. When an end point, junction, or closed figure entry point is first encountered, a *vnode* (vertex node) structure is allocated to store information about it. Border connection points are not true vertices; however, they are given vnodes until the PEs complete the topological reconstruction phase. The vnodes for a vertex has entries for its global row and column positions, its type (end, junction, border connection point, closed figure entry point, indeterminate point), a "busy" semaphore to prevent access to the vnode by more than one process at a time [Sto80], and a pointer to a list of *adjnodes* (adjacent node structures). Each adjnode gives the global row and column positions of adjacent vertices and has

a pointer to a *coordinate list* that describes the edge between the vnode point and the adjnode point. Vnodes are kept sorted by row and column index so that they can be accessed easily.

At the end of the line following phase, the graph level of the output data structure is complete for all edges that do not cross subimage boundaries. The next step, topological reconstruction, completes the graph level of the data structure.

Topological Reconstruction

In this phase, PEs exchange information about edges that cross subimage boundaries. Although reconnecting lines at subimage boundaries is an extra step in the parallel algorithm, even with serial techniques, images are frequently subdivided due to central memory restraints [Peu79]. Such a serial scheme would have the same complications as the parallel approach.

Several examples of edges processed during this phase are shown in Figure 3.7.4. Consider the edge connecting vertices A and B which is split between PE 0 and PE 1. B1 and B2 are the corresponding border connection points. The result to be obtained is to have the data structure indicate that A is adjacent to B and vice versa. However, the data structure currently indicates only the adjacency within a subimage; that is, adjacency between A and B1 in PE 0 and adjacency between B2 and B in PE 1.

To begin this phase, each PE scans its list of border connection point vnodes. Suppose PE 0 first encounters the B1 vnode. When it does, it marks the B1 vnode entry busy by setting the semaphore. This prevents other PEs from accessing B1 while the update operation is in progress. PE 0 knows that B2 is the corresponding border connection point in PE 1 because B2 appears in PE 0's augmented subimage. PE 0 sends a message to PE 1 requesting the coordinates of the vertex point adjacent to B2. In this case, PE 0 is known as the *initiating PE*. PE 0 also sends the coordinates of A, the vertex adjacent to B1, with the request as well as the address of the initiating PE (PE 0). PE 1 is interrupted upon receipt of the request and processes the message. It marks vertex B2 busy so that it does not try to initiate any queries of its own about the edge AB. PE 1 looks up B2's vnode and determines its adjacent vertex, B.

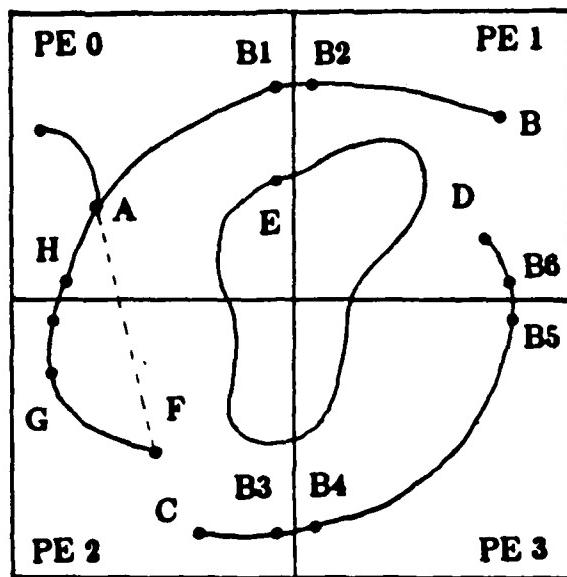


Figure 3.7.4. Connection of curves across subimage boundaries. G and H are the maximally distant points from line segment FA for PEs 2 and 0, respectively.

PE 1 then returns B's coordinates in a message to PE 0. PE 0 updates the vnode entry for A indicating that B is adjacent to it; PE 1 updates vnode B indicating that A is adjacent to it. B1 and B2 are kept as "placeholder" vnodes for later stages of the algorithm.

As an alternative to the exchange of information outlined above, PE 1 could have initiated the query with a request to PE 0. The final result would be identical.

The "locking" of the vnode entries was necessary in the example above since both PE 0 and PE 1 could have been trying to work on the same edge simultaneously. Only one PE should be allowed to connect it.

Suppose that PE 0 had initiated its request first and locked B1, but before PE 1 received the message and locked B2 due to PE 0's request, it also initiated a request (locking B2 due to its own request). In this case, PE 0 would return the message "locked" to PE 1 in response to the query about B1. Similarly, PE 1 would return the "locked" message to PE 0 because B2 is locked. To prevent *deadlock* [Sto80], that is, a situation in which each participating PE needs a resource that the other is unwilling to provide, only one PE must be allowed to continue. The situation can be resolved by having PEs unlock their vnode entries when they receive a "locked" message. Unfortunately, this results in the edge AB not being processed. The solution is to adopt a rule that prioritizes requests based on some criterion; for example, requests initiated by lower-numbered PEs are given priority. A variation on this method of resolving deadlock was discussed in [TuA83].

The edge CD in Figure 3.7.4 is processed similarly to the way described earlier. If PE 2 initiates the query, PE 3 cannot determine the other vertex (D) itself and passes the request along to PE 1. PE 1 processes the request and returns vertex D's coordinates to PE 3 which in turn returns them to PE 2. However, if PE 3 initiates the query, it has neither of the coordinates of the vertices. Thus when it is returned one vertex's coordinates, for example, "C" as a result of querying PE 2, it passes them to PE 1 when querying in the other direction. When PE 1 returns the coordinates of D, PE 2 still lacks them and must be sent an additional message. This additional message can often be avoided if PEs delay processing the "double border point" cases (such as B4-B5 in PE 3).

The closed figure in Figure 3.7.4 has no real vertices; each PE sees part of an edge (a *subedge*) with two boundary connection points. No PE knows whether its subedge is part of a closed figure or is part of an edge that crosses multiple subimage boundaries (like edge CD). Some PE queries about the vertex in one of the two directions. Eventually, the query will come back to the initiating PE which will realize that a closed figure has been found. The initiating PE designates one of its border connection points as a closed figure entry point vertex and returns the coordinates of this vertex along the path of queries.

Polygonal Approximation

The basic goal when converting images from raster to vector format is to represent curved lines (graph edges) by sets of vectors that approximate the curve shapes. Approximation algorithms require calculation of some numerical criterion that quantifies the approximation's "fit." This calculation can be computationally expensive for algorithms involving minimum perimeter polygons or least squares line fitting [Ram72]. For many applications, conversion speed is more important than making rigorously mathematical approximations and finding the absolute minimum number of vertices.

Ramer [Ram72] described an algorithm for approximating plane curves by polygons that is computationally efficient and produces near minimum numbers of vertices. The following recursive procedure is used: a curve is approximated by a straight line through its endpoints; a calculation is made to see if the approximation is acceptable and if not, the curve is broken into two curves at the point most distant from the straight-line segment. This procedure is repeated until the entire curve is approximated by a set of straight-line segments that satisfy the fit criterion.

There are various criteria for quantifying the approximation of curves by polygons [Ram72]. Here, the square of the maximum absolute distance of a curve to the straight line segment connecting its endpoints was chosen because of its computational simplicity. Furthermore, sharp spikes in an image (that may be important image information) will be preserved. Some methods of approximation such as mean squared error would not preserve this information.

One disadvantage of the polygonal approximation is that long curves require many calculations. Ramer suggested arbitrarily breaking up curves every P points (where P is perhaps 50 or 100) to minimize this problem [Ram72]. In the parallel environment, long curves are automatically segmented at every subimage boundary.

There are two ways the PEs can perform the polygonal approximations. In the first way, there is no inter-PE communication: each PE vectorizes the graph subedges within its subimage independently. Edges are arbitrarily broken at the subimage boundaries, which does not substantially affect the visual results although it may increase the total number of vectors created. If edges are generally much longer than the dimensions of a subimage, they cross multiple subimage boundaries and may be split more often than necessary. This simple approach may be acceptable for many applications. Unfortunately, the results depend on the number of PEs used to perform the algorithm, the data allocation among the PEs, and the actual image.

To obtain results comparable to the serial algorithm, the second and more complex approach must be taken. Here, PEs independently vectorize graph edges that lie entirely within their subimages and cooperate on edges that cross subimage boundaries. When processing an edge like AB in Figure 3.7.4, each PE knows the two vertex endpoints (exchanged during the topology reconstruction phase) and is responsible for determining the maximum distance between its part of the edge and the straight line segment AB. The PEs exchange their maximum distances and coordinates so each can decide where the edge should be split. Since each is executing the same algorithm, they must come to the same conclusion.

For example, consider the edge FA in Figure 3.7.4. Suppose that PE 0 determines that point H is maximally distant at 8 units and PE 2 determines that point G is maximally distant at 10 units. After the exchange of the coordinates and distances, each PE determines that point G is maximally distant from line segment FA and that the edge should be split into two vectors, FG and GA. Vector FG can now be subdivided if necessary by PE 2 on its own. However, the two PEs must cooperate again to determine if GA meets the tolerance criterion. The process begins again: PE 0 determines the maximum distance from the curve to vector GA on its side of the subimage; PE 2 does so

on its side. If after the exchange, the tolerance criterion has not been met, vector GA is subdivided further using the same procedure.

To avoid use of the deadlock resolution protocol during this phase, the exchange of maximum distance information about a given edge is initiated by the PE having the vertex point on that edge with the lower-valued row coordinate (if tie, lower-valued column coordinate). For closed figures, the PE having the closed figure entry point initiates the exchange.

The communication of the maximum distances for edge CD of Figure 3.7.4 is more complex but it mimics the protocol used in the topology reconstruction phase. Suppose that subedge C-B3 has maximum distance 10, B4-B5 has 20, and B6-D has 8. The initiating PE, PE 1, sends the distance 8 and the coordinate of the point at which it occurred to PE 3. PE 3 compares this distance to its own and passes the maximum of the two (distance 20) to PE 2. PE 2 compares the distance to its own, records the maximum (distance 20), and returns it to PE 3. PE 3 in turn returns the maximum distance to PE 1. Each now knows the maximum distance which it compares with the tolerance to determine if a split is necessary.

Communication of the distances for the closed curve of Figure 3.7.4 is similar to that described above. Each PE calculates the maximum distance of its subedge from the initiating PE's "closed curve entry point." The first split is made at the maximally-distant point.

3.7.4 Summary of Results

Results have been obtained by writing serial simulations of these parallel thinning and vectorization algorithms and by examining their performance for a number of test images. In general, these results demonstrate that parallel processing systems can be used to significantly reduce the execution times for converting images from raster to vector format. Simulations of the hybrid thinning algorithm consisting of the part-SIMD, part-MIMD distance and peeling algorithms showed improvements in average performance as compared to the distance or peeling algorithms used alone. Speedups approaching N, the number of PEs, are obtained for the vectorization algorithm so long as the subimage size does not become too small in comparison with the average length

of lines in the image. For a given machine size, arbitrarily breaking lines into vectors at PE subimage boundaries results in better speedups than those obtained for the more complex approach of iterative polygonal approximation across subimage boundaries. More detailed discussion of the parallel algorithms, including specific techniques used to match templates, follow lines, link subedges across subimage border points, and so on, are found in [KuF85].

The simulations do not model some of the subtle aspects of parallel processing such as the number of interconnection network conflicts during MIMD message passing, and the number of messages involved in resolving potential deadlock situations. The simulation programs are written in C and are found in Appendix A3.4. As the PASM prototype hardware becomes available, the data-dependent behavior of the algorithms of a complete raster to vector conversion process will be investigated and the processing of realistically-sized imagery will be undertaken.

The simulation results depend on the average sparseness/denseness of the 1-valued pixel lines in the image and on the variation in the sparseness/denseness of lines between PEs. The third column in Table 3.7.1 gives the percentage of 1-valued pixels in the PE having the fewest 1-valued pixels (sparsest subimage). The fourth column in Table 3.7.1 gives the percentage of 1-valued pixels in the PE having the most 1-valued pixels (densest subimage). From the simulation results shown in Table 3.7.1, the part-SIMD, part-MIMD algorithm enjoys a slight advantage over the all-SIMD for very sparse images (<5% 1-valued pixels) that have an evenly-distributed number of 1-valued points among the PEs (rows one and two of Table 3.7.1). The all-SIMD algorithm has an advantage for the more dense images (rows three through five of Table 3.7.1). The all-SIMD approach also enjoys an advantage for the images with disparate numbers of object pixels (last three lines of Table 3.7.1). This is expected for the cases where the densest subimage is comparatively dense (10-20% 1-valued object pixels). However, an all-MIMD algorithm would be expected to outperform the all-SIMD one where all of the subimages are rather sparse and there is not a large variation in the distribution of object pixels. The current implementation of the simulator does not allow results for the all-MIMD algorithm to be obtained.

Table 3.7.1. Thinning simulation all-SIMD and part-SIMD part-MIMD execution time results.

| N | P = N * S | % 1-valued (sparsest) | % 1-valued (densest) | all-SIMD time (seconds) | part-SIMD part-MiMD time (seconds) |
|----|------------|--------------------------|-------------------------|----------------------------|---------------------------------------|
| 16 | 256-by-256 | 1 | 1 | 1.41 | 0.97 |
| | 256-by-256 | 2 | 2 | 1.43 | 1.14 |
| | 256-by-256 | 5 | 5 | 2.88 | 2.92 |
| | 256-by-256 | 10 | 10 | 9.71 | 14.85 |
| | 256-by-256 | 25 | 25 | 46.64 | 68.04 |
| | 256-by-256 | 1 | 5 | 2.88 | 3.01 |
| | 256-by-256 | 1 | 10 | 9.71 | 14.98 |
| | 256-by-256 | 1 | 25 | 46.64 | 68.43 |

CHAPTER 8

VECTOR-TO-RASTER CONVERSION

Vector-to-raster conversion is the process of transforming a set of vectors (line segments) into a "line drawing" in an image raster. Vectors are assumed to be of the form (x_1, y_1, x_2, y_2) where x_1 and y_1 are the x- and y-coordinates of the vector starting point and x_2 and y_2 are the x- and y-coordinates of the ending point. Pixels in the image raster are referred to in the notation (x, y) where x and y are the x- and y-coordinates of the pixel. This process can be thought of as the inverse of that described in the last chapter.

Consider the vector $(0, 0, 10, 0)$ and a raster of 20-by-20 pixels. The vector-to-raster conversion algorithm would set to "1" all of the pixels "covered" by this vector. For the example, pixels $(0, 0), (1, 0), (2, 0), \dots, (10, 0)$ are covered. Since the conversion process may involve multiple vectors, some of which might "cross" each other, some pixels in the image raster will be "turned on" more than once. It is application-dependent how these cases are to be handled. If a binary image raster is to be created (e.g., for display on a monochrome bit-mapped device), the process will simply set pixels to "1." Superposition of vectors would not affect the raster or the display. On the other hand, some raster devices may allow display of varying intensities or colors for each pixel. Here the process may "increment" the pixel value by one each time a vector covers it resulting in a display that differentiates "junctions" of vectors.

Bresenham's algorithm is the one typically used to perform this transformation. Each pixel in the raster space is represented by a bit (or word) in memory. For the purposes of the algorithm, a pixel is not really a "point." Rather, it is a finite-sized "square" in the cartesian space that is labeled with the pixel coordinate. Typically in image processing, x-coordinates start with

zero and increase as one moves toward the right. Y-coordinates start with 0 and increase as one moves down. Therefore, the pixel in the upper left-hand corner is labeled (0, 0).

One of a number of conventions is used to label the "square." If the pixel coordinate is thought to be at the "center" of the "square," Bresenham's algorithm may turn on different pixels than if the pixel coordinate is assumed to be at some other location within the square, e.g., the upper left-hand corner (ULHC). Also, the notion of "covering" a pixel must be clearly defined. One convention is to assume coverage if either the vector intersects two sides of the square or it vector intersects either the ULHC or the upper right-hand corner (URHC) of the square. To illustrate these conventions, consider the vector (0, 2, 2, 0). If the "center" labeling convention is used, pixels (0, 2), (1, 2), (1, 1), (2, 1), and (2, 0) are each turned on because the vector connecting the endpoints covers these pixels (see Figure 3.8.1a). Pixels (1, 2) and (2, 1) are "covered" due to the ULHC convention. On the other hand, if the pixel labeling is assumed to be at the ULHC, (0, 2), (0, 1), (1, 1), (1, 0), and (2, 0) are turned on (Figure 3.8.1b).

Note that the convention of covering when the vector passes through only a "corner" of the square ensures that the raster formed has pixels which are 4-connected. If the convention is changed so only vectors intersecting two sides of the square is adopted, 8-connected rasters can be obtained. As an example, consider the (0, 2, 2, 0) vector. If the labeling is assumed at the center of the square, only (0, 2), (1, 1), and (2, 0) are turned on. Labeling at the ULHC results in (0, 1) and (1, 0) being turned on (neither of which is the endpoint). Most conventions result in pleasing displays, but for ease in programming the convention chosen is ULHC labeling and coverage that includes the ULHC and URHC.

Bresenham's algorithm was first coded for a serial processor in the C language for testing. Code for the algorithm appears in Appendix A3.5. The algorithm considers vectors one at a time. It begins by clipping a vector so that it begins and ends within the raster space to be initialized. This speeds processing so that the raster boundaries do not need to be checked at each iteration to follow. Next the vector's slope is calculated using the equation of the line passing through the two endpoints. The algorithm "walks" along the

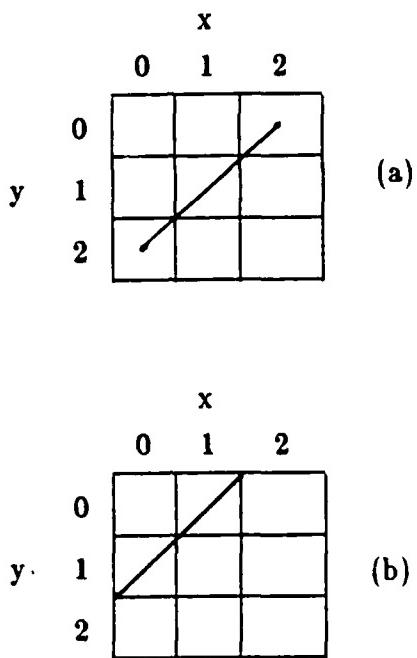


Figure 3.8.1. (a) Vector $(0, 2, 2, 0)$ with center labeling convention. (b) Vector $(0, 2, 2, 0)$ with upper left-hand corner labeling convention.

vector from beginning point to end point and determines where it intercepts each pixel's "square." For example, the vector (0, 2, 2, 0) is walked as follows. Initially, (0, 2) is turned on because of the ULHC convention. Since the slope is positive, the next pixel to be checked is either above or to the right of (0, 2). (0, 1) is found to satisfy the coverage, but (1, 2) does not. Therefore, the next pixel to be checked is either above or the right of (0, 1). (0, 0) does not satisfy the coverage, but (1, 1) does. Next, (1, 0) and (2, 1) are checked and (1, 0) found to cover. Finally (2, 0) is found to cover.

Work began by choosing a "baseline" algorithm for our studies. The "baseline" algorithm assumes the following:

1. Each PE's memory holds $1/N$ of the vectors, where N is the number of PEs.
2. Each PE's memory holds $1/N$ th of the image raster area.
3. PEs have hardware floating point capability.
4. The MIMD mode of parallelism was used.

Next, the code was re-written to simulate N PEs where $1/N$ 'th of the vectors are given to each PE to process. Assuming each PE memory will hold $1/N$ of the image raster area (a "subimage"), each simulated PE determines the set of PEs whose subimages are affected by each vector. Vectors are divided into subvectors where each subvector corresponds to the part of a vector contained within a single PE's subimage. N files of subvectors are output by the first program, one file corresponding to each PE's subimage. One way to think about this process is that Bresenham's algorithm is being applied on the "raster" of PEs to determine which PE subimages are covered by a vector.

Placing subvectors destined for different PEs in separate files allowed the number of subvectors to be counted to evaluate the severity of the "data explosion." As vectors became large in comparison to subimages and hence were split into multiple subvectors. The data explosion is important because the vectors were originally distributed among PEs randomly. Therefore, it is rather probable that a PE determining the subvectors of a vector will find that they belong to some other PE's subimage. Thus in a parallel processor, the PE would have to send the subvectors through the interconnection network. If too many PEs are used, too many subvectors are created and the network will be heavily loaded.

The second part of the program uses the N subvector files as input and applies Bresenham's algorithm to "turn on" pixels in the subimage of each of the N simulated PEs. The output is the catenation of the N subimages which form the desired picture. Other "utility" programs for manipulating the resulting pictures and displaying them on physical devices were also developed and are given in Appendix A3.5.

Simulation results from these studies indicate that the speedup for N PEs is highly data-dependent. If the average line length is much longer than the dimensions of a subimage, a large number of subvectors is created for each vector and the algorithm loses efficiency because of the "data explosion." Choosing a good "N" given the average line length is clearly important.

If there is a wide disparity in the density of vectors in a picture, the distribution of subvectors among PEs will be uneven and there will be a loss in efficiency. A load-balancing technique can be used to help solve this problem but the simulation program does not implement it. In load-balancing, a PE that received a fairly small number of subvectors can be pressed into service "helping" a heavily-loaded PE. After the lightly-loaded PE finishes its own work, it obtains subvectors from the heavily-loaded one and begins to build a new raster. In a shared-memory system, both PEs could be writing into the same raster simultaneously; however, in PASM, the local PE memories make this infeasible. At the end of processing the two PEs' local rasters can be OR-ed together to obtain the result. Clearly, only moderate to large mismatches in load make this technique feasible because entire subimages must be communicated to obtain the final result. Since the application of Bresenham's algorithm involves fairly sophisticated floating-point computations for each subvector, the expected raster update time is equivalent to the interconnection network transfer time for a fairly large number of pixels.

The overall "density" of subvectors also has an effect on whether load balancing is done or not. High subvector density implies that the cost of transmitting a subimage is relatively lower as compared to low subvector density. Low density indicates that a large number of "useless" pixels will be transmitted in the subimages that are load-balanced.

Another optimization is to vary the "shape" of the PE subimages. The subimages can be square (best for data with equal numbers of horizontal and

vertical components), have a large height/width ratio (best for data with predominately vertical components), or have a small height/width ratio (best for data with predominately horizontal components). Obviously, the smaller the subimage size, the more PEs required, but generally the less time is required of each PE.

These tradeoffs are demonstrated from statistics obtained when processing a vector drawing of the NASA Space Shuttle. The data is given in Table 3.8.1. As can be seen by this data, as the number of PEs becomes sixteen or greater, some PEs are not actively participating in Bresenham's algorithm. Also note the wide differences between the average number of subvectors/PE and the PE having the maximum number of subvectors/PE. This indicates the need for some type of load balancing scheme. Also, a data explosion exists when N becomes large, although the problem is limited due to the average vector length being short for the data used.

Continuation of research for this algorithm depends on a study of a larger number of data sets and a PASM prototype implementation. Effects not simulated are network blockages, congestion, and load-balancing. Each of these effects could have a profound influence on the execution time of the algorithm.

Table 3.8.1. Vector-to-raster conversion algorithm simulation results for 2713 vectors and a varying number of PEs.

Total vectors = 2713

| PEs | Subvectors | Subvectors/PE | | |
|--------|------------|---------------|--------|-------|
| | | (min) | (avg) | (max) |
| 262144 | 63324 | 0 | 0.24 | 21 |
| 65536 | 33446 | 0 | 0.51 | 29 |
| 16384 | 18232 | 0 | 1.11 | 46 |
| 4096 | 10500 | 0 | 2.56 | 48 |
| 1024 | 6605 | 0 | 6.45 | 110 |
| 256 | 4651 | 0 | 18.17 | 190 |
| 64 | 3686 | 0 | 57.59 | 362 |
| 16 | 3145 | 0 | 196.56 | 945 |
| 4 | 2885 | 139 | 721.25 | 1193 |
| 1 | 2713 | 2713 | 2713 | 2713 |

CHAPTER 9**SUMMARY**

This section has discussed the study of a variety of SIMD and MIMD image processing application algorithms and their execution on PASM. These algorithm studies were done to determine their implicit parallelism and therefore, how efficiently they would be executed on an SIMD/MIMD system such as PASM. Program development aids, algorithms, and simulation results were described. Performance measures for SIMD/MIMD processing were also introduced.

SUMMARY OF CONTRIBUTIONS

The three major divisions of this thesis describe an integrated and interdisciplinary contribution to the area of parallel computing research. In Part I, the architecture of PASM, an SIMD/MIMD machine, was detailed. The work demonstrated that commercial CPUs and off-the-shelf logic could be used as the basic building blocks for the PASM processing elements, SIMD control units, and other controlling processors. The mechanisms that would allow the broadcasting of instructions in SIMD mode, enabling and disabling of processing elements, and other interactions were defined. These studies have led to a modular, cost-effective design that can be rapidly prototyped. A variety of architectural elements not previously considered for PASM were discussed. These include an inter-MC network, a bi-directional inter-PE combining network, memory management hardware for relocation and protection, cache memories, paged virtual memory, and many others. Each element was analyzed to determine if its inclusion in PASM would be desirable and if not, the justifications that warrant its elimination. This part of the thesis provides an important record of the design elements of PASM that have been considered; individuals making PASM enhancements can use these elements as a starting point for more advanced designs. The analysis has also been carried forward to include the design of a PASM system with at least 1024 processing elements.

The second area of contribution has been toward the design of two languages for parallelism. The first, a PASM-prototype-specific assembly language, has been implemented and used to code a number of SIMD applications algorithms. Related utility software such as loaders, downloaders, preprocessors, and the like have also been developed or modified for use in PASM. Another contribution is the definition of a higher-level language for SIMD/MIMD parallelism based on the C programming language. It is believed

by the author to be an important contribution because it is non-PASM-specific and because it is the first to incorporate multi-dimensional parallelism and to express both SIMD and MIMD parallelism within a single language. Certain language constructs are inefficiently compiled in SIMD mode as compared to MIMD mode. The reconfigurability of PASM, in which processes can dynamically switch from SIMD to MIMD mode, can be exploited to improve performance. Here, knowledge of the design of SIMD/MIMD machines can be integrated with the design of the language. In addition, consideration has been given to the design of both the low-level kernels and the high-level PASM operating system, PASMOS. The use of a distributed memory management system with multiple file system servers is novel as is providing to each processor in the SIMD/MIMD computing engine a kernel to allow completely distributed process scheduling and local memory management.

Finally, a number of SIMD and MIMD image processing algorithms were developed and simulated. This work unifies the PASM architecture design effort and the language design effort because algorithm studies are used to drive the architecture features and because the coding of the algorithms is used to motivate the language features. Also, extensions to the conceptual modeling of performance of SIMD/MIMD machines were made. This has increased the understanding of measures such as speedup and efficiency for both SIMD and MIMD modes.

Continuation of many of the efforts outlined above is desirable. However, the areas I feel warrant the greatest immediate attention are those of the PASM operating system and parallel language. In particular, the strategies for assigning processes to processors in MIMD mode should be studied. Also, the related issue of mechanisms for inter-process synchronization and data sharing in MIMD mode needs to be examined. Each has ramifications for the design of the parallel language compiler and for the types of operating systems functions that will be provided to the programmer. It is anticipated that these studies would be based on an examination of the scheduling, sharing, and synchronization techniques employed by existing multiprocessors. These techniques need to be surveyed and evaluated for the PASM environment.

LIST OF REFERENCES

LIST OF REFERENCES

- [AdS82a] G. B. Adams III and H. J. Siegel, "On the number of permutations performable by the augmented data manipulator network," *IEEE Transactions on Computers*, Vol. C-31, April 1982, pp. 270-277.
- [AdS82b] G. B. Adams III and H. J. Siegel, "The extra stage cube: a fault-tolerant interconnection network for supersystems," *IEEE Transactions on Computers*, Vol. C-31, May 1982, pp. 443-454.
- [AdS84a] G. B. Adams III and H. J. Siegel, "A survey of fault-tolerant multistage networks and comparison to the extra stage cube," *Seventeenth Annual Hawaii International Conference on System Sciences*, January 1984, pp. 268-277.
- [AdS84b] G. B. Adams III and H. J. Siegel, "Modifications to improve the fault tolerance of the extra stage cube interconnection network," *1984 International Conference on Parallel Processing*, August 1984, pp. 169-173.
- [AdS84c] G. B. Adams III and H. J. Siegel, "The use of 4×4 switching elements in the multistage cube network," *First International Conference on Computers and Applications*, June 1984, pp. 585-592.
- [AnJ75] G. A. Anderson and E. D. Jensen, "Computer interconnection structures: taxonomy characteristics and examples," *ACM Computing Surveys*, Vol. 7, December 1975, pp. 197-213.
- [ArC75] C. Arcelli, L. Cordella, and S. Levialdi, "Parallel thinning of binary pictures," *Electronic Letters*, Vol. 11, July 1975, pp. 148-149.
- [Arn82] C. N. Arnold, "Performance evaluation of three automatic vectorizer packages," *1982 International Conference on Parallel Processing*, August 1982, pp. 235-242.
- [BaB68] G. H. Barnes, R. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The Illiac IV computer," *IEEE Transactions on Computers*, Vol. C-17, August 1968, pp. 746-757.

- [Bar82] J. G. P. Barnes, *Programming in Ada*, Addison-Wesley, London, England, 1982.
- [Bat74] K. E. Batcher, "STARAN parallel processor system hardware," *AFIPS 1974 National Computer Conference*, May 1974, pp. 405-410.
- [Bat76] K. E. Batcher, "The flip network in STARAN," *1976 International Conference on Parallel Processing*, August 1976, pp. 65-71.
- [Bat77a] K. E. Batcher, "The multidimensional access memory in STARAN," *IEEE Transactions on Computers*, Vol. C-26, February 1977, pp. 174-177.
- [Bat77b] K. E. Batcher, "STARAN series E," *1977 International Conference on Parallel Processing*, August 1977, pp. 140-143.
- [Bat82] K. E. Batcher, "Bit serial parallel processing systems," *IEEE Transactions on Computers*, Vol. C-31, May 1982, pp. 377-384.
- [Bat84] K. E. Batcher, "The MPP Staging Memory," *1984 International Conference on Parallel Processing*, August 1984, pp. 496-498.
- [Bau74] L. H. Bauer, "Implementation of data manipulating functions on the STARAN associative processor," *1974 Sagamore Computer Conference on Parallel Processing*, August 1974, pp. 209-227.
- [BoD72] W. J. Bouknight, S. A. Denenberg, D. E. McIntryre, J. M. Randall, A. H. Sameh, and D. L. Slotnick, "The ILLIAC IV system," *Proceedings of the IEEE*, Vol. 60, April 1972, pp. 369-388.
- [Bur77] Burroughs Corporation, *BSP-Burroughs Scientific Processor*, Burroughs Corporation, 1977.
- [Sei85] C. L. Seitz, "The Cosmic Cube," *Communications of the ACM*, January 1985, pp. 22-33.
- [CIS83] C. Cline and H. J. Siegel, "Extensions of Ada for SIMD parallel processing," *IEEE Computer Society Seventh International Computer Software and Applications Conference (COMPSAC)*, November 1983, pp. 366-372.
- [CIS84] C. Cline and H. J. Siegel, "A comparison of parallel language approaches to data representation and data transferral," *Computer Data Engineering Conference (COMPDEC)*, April 1984, pp. 60-66.
- [CIS85] C. L. Cline and H. J. Siegel, "Augmenting Ada for SIMD parallel processing," *IEEE Transactions on Software Engineering*, Vol. SE-11, September 1985, pp. 970-977.

- [Com84] D. Comer, *Operating System Design, the XINU Approach*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.
- [Cor72] J. A. Cornell, "Parallel processing of ballistic missile defense radar data with PEPE," *IEEE Computer Society Compcon 72*, September 1972, pp. 69-72.
- [CrG72] B. A. Crane, M. J. Gilmartin, J. H. Huttenhoff, P. T. Rux, and R. R. Shively, "PEPE computer architecture," *IEEE Computer Society Compcon 72*, September 1972, pp. 57-60.
- [CrG85] W. Crowther, J. Goodhue, R. Thomas, W. Milliken, and T. Blackadar, "Performance measurements on a 128-node butterfly parallel processor," *1985 International Conference on Parallel Processing*, August 1985, pp. 531-540.
- [DaH85a] N. J. Davis IV, W. T. Y. Hsu, and H. J. Siegel, "Fault location in distributed control interconnection networks," *1985 International Conference on Parallel Processing*, August 1985, pp. 403-410.
- [DaH85b] N. J. Davis IV, W. T.-Y. Hsu, and H. J. Siegel, "Fault location techniques for distributed control interconnection networks," *IEEE Transactions on Computers*, October 1985, pp. 902-910.
- [DaO85] N. J. Davis IV, J. Ott, H. J. Siegel, and A. L. Overvig, *Simulation Studies of the Generalized Cube Interconnection Network*, Technical Report, School of Electrical Engineering, Purdue University, to appear, 1985.
- [DaS85a] N. J. Davis IV and H. J. Siegel, "The PASM prototype interconnection network," *1985 National Computer Conference*, July 1985, pp. 183-190.
- [DaS85b] N. J. Davis IV and H. J. Siegel, "The performance analysis of partitioned circuit switched multistage interconnection networks," *Twelfth Annual Symposium on Computer Architecture*, June 1985, pp. 387-394.
- [Dav74] E. W. Davis, "STARAN parallel processor system software," *AFIPS 1974 National Computer Conference*, May 1974, pp. 17-22.
- [Den70] P. J. Denning, "Virtual memory," *ACM Computing Surveys*, Vol. 2, September 1970, pp. 153-188.
- [DiK85] H. Dietz and D. Klappholz, "Refined C: a sequential language for parallel processing," *1985 International Conference on Parallel Processing*, August 1985, pp. 442-449.

- [Dij68] E. W. Dijkstra, "Cooperating sequential processes," in *Programming Languages*, F. Genuys, ed., Academic Press, New York, NY, 1968, pp. 43-112.
- [DuH73] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, John Wiley and Sons, New York, NY, 1973.
- [Duf76] M. J. B. Duff, "CLIP 4: a large scale integrated circuit array parallel processor," *Third International Conference on Pattern Recognition*, 1976, pp. 728-732.
- [Ens74] P. H. Enslow, Jr., "Appendix A: Parallel Element Processing Ensemble PEPE," in *Multiprocessors and Parallel Processing*, John Wiley and Sons, New York, NY, 1974, pp. 139-149.
- [Fal76] H. Falk, "Reaching for a gigaflop," *IEEE Spectrum*, October 1976, pp. 65-69.
- [Fen74] T. Y. Feng, "Data manipulating functions in parallel processors and their implementations," *IEEE Transactions on Computers*, Vol. C-23, March 1974, pp. 309-318.
- [Fer82] C. Fernstrom, "Programming techniques on the LUCAS associative array computer," *1982 International Conference on Parallel Processing*, August 1982, pp. 253-261.
- [Fly86] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, Vol. 54, December 1966, pp. 1901-1909.
- [FoE84] L. Foti, D. English, R. P. Hopkins, D. J. Kinniment, P. C. Treleaven, and W. L. Wang, "Reduced-instruction set multi-microcomputer system," *AFIPS 1984 National Computer Conference*, June 1984, pp. 69-76.
- [Fou81] T. J. Fountain, "CLIP4: progress report," in *Languages and Architectures for Image Processing*, M. J. B. Duff and S. Levialdi, eds., Academic Press, London, England, 1981, pp. 281-291.
- [Fre61] H. Freeman, "Techniques for the digital computer analysis of chain-encoded arbitrary plane curves," *Proc. NEC*, Vol. 17, October 1961, pp. 421-432.
- [GoL73] G. R. Goke and G. J. Lipovski, "Banyan networks for partitioning multiprocessor systems," *First Annual Symposium on Computer Architecture*, December 1973, pp. 21-28.
- [GoG83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer -- designing an MIMD shared-memory parallel computer," *IEEE Transactions on Computers*, Vol. C-32, February 1983, pp. 175-189.

- [Gri74] R. Grishman, *Assembly Language Programming for the Control Data 6000 Series and the Cyber 70 Series*, Algorithmics Press, New York, NY, 1974.
- [Hil83] C. J. Hilditch, "Comparison of thinning algorithms on a parallel processor," *Image and Vision Processing*, Vol. 1, August 1983, pp. 115-132.
- [HwB84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, NY, 1984.
- [Inm83] Inmos Corporation, *Occam*, Inmos Corporation, Colorado Springs, CO, 1983.
- [Joh78] S. C. Johnson, *Lint, a C Program Checker*, Computer Science Technical Report, Bell Laboratories, 1978.
- [Jor83] H. F. Jordan, "Performance measurement of HEP - a pipelined MIMD computer," *Tenth Annual Symposium on Computer Architecture*, June 1983, pp. 207-212.
- [KaK78] S. I. Kartashev and S. P. Kartashev, "Dynamic architectures: problems and solutions," *Computer*, July 1978, pp. 26-42.
- [KaK79] S. I. Kartashev and S. P. Kartashev, "A multicomputer system with dynamic architecture," *IEEE Transactions on Computers*, Vol. C-28, October 1979, pp. 704-720.
- [KaK80] S. I. Kartashev and S. P. Kartashev, "Problems of designing super-systems with dynamic architecture," *IEEE Transactions on Computers*, Vol. C-29, December 1980, pp. 1114-1132.
- [Ker79] B. W. Kernighan, *UNIX Time-Sharing System: Unix Programmer's Manual (Seventh Edition)*, Bell Laboratories, Murray Hill, NJ, 1979.
- [KeR78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
- [Kry76] A. J. Krygiel, "An implementation of the Hadamard transform on the STARAN associative array processor," *1976 International Conference on Parallel Processing*, August 1976, pp. 34.
- [KuS82] D. J. Kuck and R. A. Stokes, "The Burroughs Scientific Processor (BSP)," *IEEE Transactions on Computers*, Vol. C-31, May 1982, pp. 363-376.
- [Kue81] J. T. Kuehn, *Emulation of SIMD Machine Architectures*, MSEE thesis, School of Electrical Engineering, Purdue University, 1981, 257 pp.

- [Kue84a] J. T. Kuehn, *Parallel C Language Extensions for PASM*, Internal report, unpublished, 1984.
- [Kue84b] J. T. Kuehn, *PA68 Assembler Reference Manual*, Internal report, unpublished, 1984.
- [KuF85] J. T. Kuehn, J. A. Fessler, and H. J. Siegel, "Parallel image thinning and vectorization on PASM," *1985 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, June 1985, pp. 368-374.
- [KuS85] J. T. Kuehn, T. Schwederski, and H. J. Siegel, "Design of a 1024-processor PASM system," *First International Conference on Supercomputing Systems*, December 1985, pp. 603-612.
- [KuS81] J. T. Kuehn and H. J. Siegel, "Simulation studies of PASM in SIMD mode," *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, November 1981, pp. 43-50.
- [KuS84] J. T. Kuehn and H. J. Siegel, "Simulation studies of a parallel histogramming algorithm for PASM," *Seventh International Conference on Pattern Recognition*, July 1984, pp. 646-649.
- [KuS85] J. T. Kuehn and H. J. Siegel, "Extensions to the C programming language for SIMD/MIMD parallelism," *1985 International Conference on Parallel Processing*, August 1985, pp. 232-235.
- [KuS86a] J. T. Kuehn and H. J. Siegel, "Multifunction processing with PASM," in *Intermediate-level Image Processing*, M. J. B. Duff, ed., Academic Press, Inc., London, England, to appear, 1986.
- [KuS86b] J. T. Kuehn and H. J. Siegel, "Simulation based performance measures for SIMD/MIMD processing," in *Computing Structures and Image Processing*, K. Preston, Jr., L. Uhr, M. J. B. Duff, and S. Levialdi, eds., Academic Press, San Diego, CA, to appear, 1986.
- [KuS83] J. T. Kuehn, H. J. Siegel, and M. J. Grosz, "A distributed memory management system for PASM," *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, October 1983, pp. 101-108.
- [KuS82] J. T. Kuehn, H. J. Siegel, and P. D. Hallenbeck, "Design and simulation of an MC68000-based multimicroprocessor system," *1982 International Conference on Parallel Processing*, August 1982, pp. 353-362.

- [KuS85] J. T. Kuehn, H. J. Siegel, D. L. Tuomenoksa, and G. B. Adams III, "The use and design of PASM," in *Integrated Technology for Parallel Image Processing*, S. Levialdi, ed., Academic Press, San Diego, CA, 1985, pp. 133-152.
- [Kun82] H. T. Kung, "Why systolic architectures?", *Computer*, Vol. 15, January 1982, pp. 37-46.
- [KuW81] T. Kushner, A. Y. Wu, and A. Rosenfeld, "Image processing in ZMOB," *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, November 1981, pp. 88-95.
- [KuW82] T. Kushner, A. Y. Wu, and A. Rosenfeld, "Image processing on ZMOB," *IEEE Transactions on Computers*, Vol. C-31, October 1982, pp. 943-951.
- [LaS76] T. Lang and H. S. Stone, "A shuffle-exchange network v. h simplified control," *IEEE Transactions on Computers*, Vol. C-25, January 1976, pp. 55-66.
- [Law75] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Transactions on Computers*, Vol. C-24, December 1975, pp. 1145-1155.
- [Led83] H. Ledgard, *Ada, An Introduction*, Springer-Verlag, New York, NY, 1983.
- [Li84] K-C. Li, *Vector C - a Programming Language for Vector Processing*, Ph.D. Dissertation, Department of Computer Science, Purdue University, 1984.
- [LiS85] K-C. Li and H. Schwetman, "Vector C: a vector processing language," *Journal of Parallel and Distributed Computing*, Vol. 2, February 1985, pp. 132-169.
- [LiT77] G. J. Lipovski and A. R. Tripathi, "A reconfigurable varistructure array processor," *1977 International Conference on Parallel Processing*, August 1977, pp. 165-174.
- [Duf85] M.J.B. Duff, "Real Applications on CLIP4," in *Integrated Technology for Parallel Image Processing*, Academic Press, Orlando, Florida, 1985, pp. 153-165.
- [MaC82] M. Malek, S. Cheemalavagu, M. S. Juang, and D. B. Rathi, *Design, Packaging, Performance and Self-diagnosis of a 4x4 Banyan Interconnection Network*, Department of Electrical Engineering, The University of Texas at Austin, 1982.

- [McA81] R. J. McMillen, G. B. Adams III, and H. J. Siegel, "Performance and implementation of 4x4 switching nodes in an interconnection network for PASM," *1981 International Conference on Parallel Processing*, August 1981, pp. 229-233.
- [McS82a] R. J. McMillen and H. J. Siegel, "Performance and fault tolerance improvements in the Inverse Augmented Data Manipulator network," *Ninth Annual Symposium on Computer Architecture*, April 1982, pp. 63-72.
- [McS82b] R. J. McMillen and H. J. Siegel, "A comparison of cube type and data manipulator type networks," *Third International Conference on Distributed Computing Systems*, October 1982, pp. 614-621.
- [MeS85] D. G. Meyer, H. J. Siegel, T. Schwederski, N. J. Davis IV, and J. T. Kuehn, "The PASM parallel system prototype," *IEEE Computer Society Spring Compcon 85*, February 1985, pp. 429-434.
- [Mil79] D. L. Milgram, "Region extraction using convergent evidence," *Computer Graphics and Image Processing*, Vol. 11, 1979, pp. 1-12.
- [MiR81] O. R. Mitchell, A. P. Reeves, and K-S. Fu, "Shape and texture measurements for automated cartography," *1981 IEEE Computer Society Conference on Pattern Recognition and Image Processing*, August 1981, pp. 367.
- [MoM81] Mostek Corporation, Motorola Inc., and Signetics/Philips, *VME Bus Specification Manual*, Revision A, Mostek Corporation, Motorola Inc., and Signetics/Philips, 1981.
- [Mot84a] Motorola, Inc., *M68000 16/32-Bit Microprocessor Programmer's Reference Manual (fourth edition)*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
- [Mot84b] Motorola, Inc., *MC68020 32-Bit Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
- [Mot85] Motorola, Inc., *MC68881 Floating Point Coprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, NJ, 1985.
- [MuD82] T. N. Mudge, E. J. Delp, L. J. Siegel, and H. J. Siegel, "Image coding using the multimicroprocessor system PASM," *IEEE Computer Society Conference on Pattern Recognition and Image Processing*, June 1982, pp. 200-205.
- [Nae84] R. Naeini, "A few statement types adapt C language to parallel processing," *Electronics*, June 28 1984, pp. 125-129.

- [Nut77a] G. J. Nutt, "Microprocessor implementation of a parallel processor," *Fourth Annual Symposium on Computer Architecture*, March 1977, pp. 147-152.
- [Nut77b] G. J. Nutt, "A parallel processor operating system comparison," *IEEE Transactions on Software Engineering*, Vol. SE-3, November 1977, pp. 467-475.
- [Pat81] J. H. Patel, "Performance of processor-memory interconnections for multiprocessors," *IEEE Transactions on Computers*, Vol. C-30, October 1981, pp. 771-780.
- [PaP82] D. A. Patterson and R. S. Piepho, "Assessing RISCs in high-level language support," *IEEE Computer*, Vol. 15, November 1982, pp. 9-19.
- [PaS82] D. A. Patterson and C. H. Sequin, "A VLSI RISC," *IEEE Computer*, Vol. 15, September 1982, pp. 8-21.
- [Pea77] M. C. Pease III, "The indirect binary n-cube microprocessor array," *IEEE Transactions on Computers*, Vol. C-26, May 1977, pp. 458-473.
- [Per79] R. H. Perrott, "A language for array and vector processors," *ACM Transactions on Programming Languages and Systems*, Vol. 1, October 1979, pp. 177-195.
- [PeC85] R. H. Perrott, D. Crookes, P. Milligan, and W. R. M. Purdy, "A compiler for an array and vector processing language," *IEEE Transactions on Software Engineering*, Vol. SE-11, May 1985, pp. 471-478.
- [Peu81] D. Peuquet, "An examination of techniques for reformatting digital cartographic data. Part 1: The raster-to-vector process," *Cartographica*, Vol. 18, 1981, pp. 34-38.
- [Peu79] D. J. Peuquet, "Raster processing: an alternative approach to automated cartographic data handling," *The American Cartographer*, Vol. 6, October 1979, pp. 129-139.
- [PrK80] D. K. Pradhan and K. L. Kodandapani, "A uniform representation of single- and multistage interconnection networks used in SIMD machines," *IEEE Transactions on Computers*, Vol. C-29, September 1980, pp. 777-791.
- [RaL77] C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture," *Computing Surveys*, Vol. 9, March 1977, pp. 61-102.

- [Ram72] U. Ramer, "An iterative procedure for the polygonal approximation of plane curves," *Computer Graphics and Image Processing*, Vol. 1, 1972, pp. 244-256.
- [Ret83] R. D. Rettberg, *Development of a Voice Funnel System*, Quarterly Technical Report No. 17, Bolt, Beranek, and Newman, Inc., 1983.
- [RiJ85] T. A. Rice and L. H. Jamieson, "Parallel processing for computer vision," in *Integrated Technology for Parallel Image Processing*, S. Levialdi, ed., Academic Press, London, England, 1985, pp. 57-79.
- [RiS83a] T. A. Rice and L. J. Siegel, "Parallel processing for computationally intensive speech analysis operations," *1983 International Conference on Acoustics, Speech, and Signal Processing*, April 1983, pp. 471-474.
- [RiS83b] T. A. Rice and L. J. Siegel, "Parallel algorithms for computer vision," *Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, October 1983, pp. 93-100.
- [RoP77] D. Rohrbacher and J. L. Potter, "Image processing with the STARAN parallel computer," *Computer*, Vol. 10, August 1977, pp. 54-59.
- [RoP66] A. Rosenfeld and J. L. Pfaltz, "Sequential operations in digital picture processing," *Journal of the Association for Computing Machinery*, Vol. 13, October 1966, pp. 471-494.
- [Rum77] J. Rumbaugh, "A data flow multiprocessor," *IEEE Transactions on Computers*, Vol. C-26, February 1977, pp. 138-146.
- [SeU80] M. C. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski, "An overview of the Texas Reconfigurable Array Computer," *AFIPS 1980 National Computer Conference*, June 1980, pp. 631-641.
- [Sie77a] H. J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," *IEEE Transactions on Computers*, Vol. C-26, February 1977, pp. 153-161.
- [Sie77b] H. J. Siegel, "Controlling the active/inactive status of SIMD machine processors," *1977 International Conference on Parallel Processing*, August 1977, pp. 183.
- [Sie79] H. J. Siegel, "Interconnection networks for SIMD machines," *Computer*, Vol. 12, June 1979, pp. 57-65.
- [Sie80] H. J. Siegel, "The theory underlying the partitioning of permutation networks," *IEEE Transactions on Computers*, Vol. C-29, September 1980, pp. 791-801.

- [Sie85] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, Lexington Books, D. C. Heath and Co., Lexington, MA, 1985.
- [SiK79] H. J. Siegel, F. C. Kemmerer, and M. Washburn, "Parallel memory system for a partitionable SIMD/MIMD machine," *1979 International Conference on Parallel Processing*, August 1979, pp. 212-221.
- [SiK80] H. J. Siegel and J. T. Kuehn, *Parallel Image Processing / Feature Extraction Algorithms and Architecture Emulation: Interim Report for Fiscal 1980, Volume II: Architecture Emulation*, Technical Report TR-EE 80-58, School of Electrical Engineering, Purdue University, 1980, 221 pp.
- [SiK81] H. J. Siegel and J. T. Kuehn, *Parallel Image Processing / Feature Extraction Algorithms and Architecture Emulation: Interim Report for Fiscal 1981, Volume II: Architecture Emulation*, Technical Report TR-EE 81-36, School of Electrical Engineering, Purdue University, 1981.
- [SiK82] H. J. Siegel and J. T. Kuehn, *Design and Simulation of a Multimicroprocessor System for Mapping Applications*, Technical Report TR-EE 83-18, School of Electrical Engineering, Purdue University, 1982, 334 pp.
- [SiK85] H. J. Siegel and J. T. Kuehn, "PASM: a partitionable SIMD/MIMD system for parallel image processing research," in *Algorithmically Specialized Parallel Computers*, L. Snyder, L. H. Jamieson, D. B. Gannon, and H. J. Siegel, eds., Academic Press, New York, NY, 1985, pp. 69-78.
- [SiM80] H. J. Siegel and R. J. McMillen, *The Use of the Multistage Cube Network in a Multimicroprocessor Test Bed*, Technical Report TR-EE 80-16, School of Electrical Engineering, Purdue University, 1980, 75 pp.
- [SiM81a] H. J. Siegel and R. J. McMillen, "Using the augmented data manipulator network in PASM," *Computer*, Vol. 14, February 1981, pp. 25-33.
- [SiM81b] H. J. Siegel and R. J. McMillen, "The cube network as a distributed processing test bed switch," *Second International Conference on Distributed Computing Systems*, April 1981, pp. 377-387.
- [SiM81c] H. J. Siegel and R. J. McMillen, "The multistage cube: a versatile interconnection network," *Computer*, Vol. 14, December 1981, pp. 65-76.

- [SiM78a] H. J. Siegel and P. T. Mueller, Jr., "The organization and language design of microprocessors for an SIMD/MIMD system," *Second Rocky Mountain Symposium on Microcomputers*, August 1978, pp. 311-340.
- [SiM78b] H. J. Siegel, P. T. Mueller, Jr., and H. E. Smalley, Jr., "Control of a partitionable multimicroprocessor system," *1978 International Conference on Parallel Processing*, August 1978, pp. 9-17.
- [SiS84] H. J. Siegel, T. Schwederski, N. J. Davis IV, and J. T. Kuehn, "PASM: a reconfigurable parallel system for image processing," *Workshop on Algorithm-guided Parallel Architectures for Automatic Target Recognition*, July 1984, pp. 263-291. (Also appears in the ACM SIGARCH newsletter: *Computer Architecture News*, Vol. 12, No. 4, September 1984, pp. 7-19).
- [SiS81b] H. J. Siegel, I. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Transactions on Computers*, Vol. C-30, December 1981, pp. 934-947.
- [SiS81b] H. J. Siegel and P. H. Swain, "Contextual classification on PASM," *IEEE Computer Society Conference on Pattern Recognition and Image Processing*, August 1981, pp. 320-325.
- [SiS82] H. J. Siegel, P. H. Swain, and B. W. Smith, "Remote sensing on PASM and CDC Flexible Processors," in *Multicomputers and Image Processing: Algorithms and Programs*, K. Preston and L. Uhr, eds., Academic Press, New York, NY, 1982, pp. 331-342.
- [Sie81] L. J. Siegel, "Image processing on a partitionable SIMD machine," in *Languages and Architectures for Image Processing*, M. J. B. Duff and S. Levialdi, eds., Academic Press, London, England, 1981, pp. 293-300.
- [SiD81] L. J. Siegel, E. J. Delp, T. N. Mudge, and H. J. Siegel, "Block truncation coding on PASM," *Nineteenth Allerton Conference on Communication, Control, and Computing*, October 1981, pp. 891-900.
- [SiS82a] L. J. Siegel, H. J. Siegel, and A. E. Feather, "Parallel processing approaches to image correlation," *IEEE Transactions on Computers*, Vol. C-31, March 1982, pp. 208-218.
- [SiS82b] L. J. Siegel, H. J. Siegel, and P. H. Swain, "Performance measures for evaluating algorithms for SIMD machines," *IEEE Transactions on Software Engineering*, Vol. SE-8, July 1982, pp. 319-331.

- [SiS82c] L. J. Siegel, H. J. Siegel, and P. H. Swain, "Parallel algorithm performance measures," in *Multicomputers and Image Processing: Algorithms and Programs*, K. Preston and L. Uhr, eds., Academic Press, New York, NY, 1982, pp. 241-252.
- [SiS83] L. J. Siegel, H. J. Siegel, P. H. Swain, G. B. Adams III, G-M. Lin, D. L. Tuomenoksa, and T. A. Rice, *Parallel Processing Approaches to Production Scenarios for Mapping Applications*, TR-EE 83-27, Purdue University, School of Electrical Engineering, 1983, 265 pp.
- [SlB62] D. L. Slotnick, W. C. Borck, and R. C. McReynolds, "The SOLOMON Computer," *1962 AFIPS Fall Joint Computer Conference*, 1962, pp. 97-107.
- [Ste75] K. G. Stevens, "CFD - A FORTRAN-like language for the Illiac IV," *ACM Conference on Programming Languages and Compilers for Parallel and Vector Machines*, March 1975, pp. 72-76.
- [Sto71] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Transactions on Computers*, Vol. C-20, February 1971, pp. 153-161.
- [Sto80] H. S. Stone, "Parallel computers," in *Introduction to Computer Architecture (second edition)*, H. S. Stone, ed., Science Research Associates, Inc., Chicago, IL, 1980, pp. 363-425.
- [SuR82] R. E. Suciu and A. P. Reeves, "A comparison of differential and moment based edge detectors," *1982 IEEE Computer Society Conference on Pattern Recognition and Image Processing*, June 1982, pp. 97-102.
- [SwT81] P. H. Swain, J. C. Tilton, and S. B. Vardeman, "Contextual classification of multispectral image data," *Pattern Recognition*, Vol. 13, 1981, pp. 429-441.
- [SwB77] R. J. Swan, A. Bechtolsheim, K. W. Lai, and J. K. Ousterhout, "The implementation of the Cm* multimicroprocessor," *AFIPS 1977 National Computer Conference*, June 1977, pp. 645-655.
- [SwF77] R. J. Swan, S. Fuller, and D. P. Siewiorek, "Cm*: a modular multimicroprocessor," *AFIPS 1977 National Computer Conference*, June 1977, pp. 637-644.
- [Tan81a] A. S. Tanenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [Tan81b] S. L. Tanimoto, *Towards Hierarchical Cellular Logic: Design Considerations for Pyramid Machines*, Technical Report 81-02-01, Computer Science Department, University of Washington, 1981.

- [ThW75] K. J. Thurber and L. D. Wald, "Associative and parallel processors," *ACM Computing Surveys*, Vol. 7, December 1975, pp. 215-255.
- [TrB82] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, "Data-driven and demand-driven computer architecture," *Computing Surveys*, Vol. 14, March 1982, pp. 93-143.
- [TuP85] J. Tuazon, J. Peterson, M. Pniel, and D. Liberman, "Caltech/JPL Mark II hypercube concurrent processor," *1985 International Conference on Parallel Processing*, August 1985, pp. 666-673.
- [TuA83] D. L. Tuomenoksa, G. B. Adams III, H. J. Siegel, and O. R. Mitchell, "A parallel algorithm for contour extraction: advantages and architectural implications," *1983 IEEE Computer Society Symposium on Computer Vision and Pattern Recognition*, June 1983, pp. 336-344.
- [TuS81] D. L. Tuomenoksa and H. J. Siegel, "Application of two-dimensional bin packing algorithms for task scheduling in the PASM multimicrocomputer system," *Nineteenth Allerton Conference on Communication, Control, and Computing*, October 1981, pp. 542.
- [TuS82a] D. L. Tuomenoksa and H. J. Siegel, "Analysis of the PASM control system memory hierarchy," *1982 International Conference on Parallel Processing*, August 1982, pp. 363-370.
- [TuS82b] D. L. Tuomenoksa and H. J. Siegel, "Analysis of multiple-queue task scheduling algorithms for multiple-SIMD machines," *Third International Conference on Distributed Computing Systems*, October 1982, pp. 114-121.
- [TuS83] D. L. Tuomenoksa and H. J. Siegel, "Preloading schemes for the PASM parallel memory system," *1983 International Conference on Parallel Processing*, August 1983, pp. 407-415.
- [TuS84a] D. L. Tuomenoksa and H. J. Siegel, "A distributed operating system for PASM," *17th Hawaii International Conference on System Sciences*, January 1984, pp. 89-77.
- [TuS84b] D. L. Tuomenoksa and H. J. Siegel, "Task preloading schemes for reconfigurable parallel processing systems," *IEEE Transactions on Computers*, Vol. C-33, October 1984, pp. 895-905.
- [TuS85] D. L. Tuomenoksa and H. J. Siegel, "Task scheduling on the PASM parallel processing system," *IEEE Transactions on Software Engineering*, Vol. SE-11, February 1985, pp. 145-157.

- [Uhr81] L. Uhr, "Converging pyramids of arrays," *Proceedings of the Workshop on Computer Architecture for Pattern Analysis and Image Data Base Management*, November 1981, pp. 31-34.
- [Uhr83] L. Uhr, "Pyramid multi-computer structures, and augmented pyramids," in *Computing Structures for Image Processing*, M. J. B Duff, ed., Academic Press, London, 1983, pp. 95-112.
- [Ber83] University of California - Berkeley, *UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California - Berkeley, Berkeley, CA, 1983.
- [ViC78] C. R. Vick and J. A. Cornell, "PEPE architecture - present and future," *AFIPS 1978 National Computer Conference*, June 1978, pp. 981-992.
- [VoF77] J. M. Vocar and R. O. Faiss, "Warp processing using STARAN," *1977 Conference on Picture Data Description and Management*, April 1977, pp. 68-76.
- [WaS82] M. R. Warpenburg and L. J. Siegel, "SIMD image resampling," *IEEE Transactions on Computers*, Vol. C-31, October 1982, pp. 934-942.
- [Wil72] D. E. Wilson, "The PEPE support software system," *IEEE Computer Society Comcon 72*, September 1972, pp. 61-64.
- [WuF80] C-L. Wu and T. Y. Feng, "On a class of multistage interconnection networks," *IEEE Transactions on Computers*, Vol. C-29, August 1980, pp. 694-702.
- [WuB72] W. Wulf and C. Bell, "C.mmp--A multi-miniprocessor," *AFIPS 1972 Fall Joint Computer Conference*, December 1972, pp. 765-777.
- [WuC74] W. Wulf, E. Cohen, W. Corwin, A. K. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: the kernel of a multiprocessor operating system," *Communications of the ACM*, Vol. 17, June 1974, pp. 337-345.
- [YaF77] S. S. Yau and H. S. Fung, "Associative processor architecture - a survey," *Computing Surveys*, Vol. 9, March 1977, pp. 3-27.